Neural Networks as Components

G. Smith
guy@minster.york.ac.uk

J. Austin
austin@minster.york.ac.uk

Advanced Computer Architecture Group
Department of Computer Science
University of York
York UK

1 Abstract

This paper promotes and examines the use of neural networks as components. The properties of components desirable for constructing large flexible[1] systems are listed. Some examples of such systems are examined. Neural networks are shown to have these desirable properties. Those characteristics of individual neural network algorithms relevent to system design are described, and then examined for some well–known algorithms. The process of designing a neural network system for a particular task is given as an example. Finally, directions for future research are given.

2 Introduction

Most current neural network research is in one of three areas:

Performance improving the performance of existing algorithms or developing new algorithms,

Applications using a single neural network to solve real world problems to which they are well suited, such as low level vision and speech tasks, modeling non–linear processes, controlling robot arms, and finding good solutions to combinatorial problems,

Biology modeling the behaviour of real neural systems as a tool in neurobiology and psychology.

It is the purpose of this paper to promote a fourth area of research:

Components investigating the properties of neural networks as components of larger systems.

Little research There are several reasons there has been little research on incorporating neural networks into larger systems.

Much of the research has centred on understanding the properties of individual neural network algorithms. This research was necessary before neural networks could be incorporated into larger systems, since one can not confidently design a system if its components are not well understood. However, an understanding of the properties of neural networks, both in general and specific algorithms, has been reached. Though this understanding can be further refined, it is sufficient for the design of systems of neural networks.

Also, many hoped that neural network algorithms would be able to solve any problem. If a single algorithm would solve a problem, why design a system to do it? Experience has shown neural network algorithms do not scale up easily. They are very good at solving small problems. When presented with large problems, their demands in terms of training set size, training time and network size make them infeasible.

Existing research Some research is already being done on the incorparation of neural networks into larger systems.

Pomerleau [Pom91] [Pom92] uses back propagation nets in a car driving system. Kuperstein [Kup90] uses neural networks for both the vision and arm movements of a robot.

[1] A flexible system is one in which simple changes of functionality and performance can be implemented without major redesign.

Nguyen and Widrow [NW89] use two interacting networks. One network learns to model a reversing articulated truck, the other learns to steer the truck, using the model to learn with.

Munro [Mun87] uses two interacting back propagation networks to solve problems where only scalar measures of a trial outputs, but not target vectors, are available. One of the networks, LowNet, learns the desired mapping. The other network, HighNet, learns to predict the scalar measure from LowNet's inputs and outputs. The back propagated error of its estimate is used as an error vector for LowNet.

Jacobs et al. [Jac90] and Jacobs [Jac91] use two layers of backpropagation networks. In the first layer, there are several networks, all learning the same task. The second layer chooses between the outputs of the first layer's networks. As it learns what regions of the input space each of the first layer networks performs best in, it forces the first layer networks to specialise in particular regions of the input space.

This paper is intended to promote a new view of neural networks, not as a review or comparison of existing algorithms. It is necessary to mention and compare many well-known algorithms. These comparisons are not intended to be definitive, but are provided as examples to demonstrate the main idea of the paper. There are doubtless variations of each of the algorithms mentioned that fall outside the description given here.

3 Properties of Components

Large flexible systems are not monolithically structured. They are built of components. For a system to be robust in the presence of external noise, internal faults, and for the system to be flexible enough perform a range of tasks and to expand and contract, the components need to have a well defined and consistent interface, to be tolerant of noisy input, and to be available with a broad range of functionality and performance.

The desirable properties of components are listed here.

Interface The interface to components should be consistent over all components, and should not reflect the complexity of the operation performed by that component.

Tolerance If the entire system is to be robust in the presence of noise, the behaviour of individual components should vary little with noise.

Range of functionality Different parts of a system perform different functions. Though components must communicate through a common interface, they need to perform different functions.

Range of performance If components with the same functionality, but different performance, are available, it enables the performance of the entire system to be altered without redesigning the system. Also, different parts of a system may require the same functionality, but with a different emphasis on performance. As an example, the cache and main memory components of a computer have different trade–offs between speed and cost requirements.

How the properties of a system depend on the properties of its components can be clarified by examples. Computers and the InterNet are good examples of large flexible systems.

Computers are built of components at many level of abstraction: boards, chips and transistors. Let us consider chips as components. The physical interface between chips is well defined in terms of voltage levels on wires and clock cycles. Also, the logical structure of the interface is well defined by accepted formats for floating point, integer and character data. Chips are available for many functions: memory, CPU, floating point accelerators, etc. These chips are available at different speeds, and with varying tolerances to temperature and radiation. Because there is a consistent interface between chips, the performance of the entire system (the computer) can be enhanced, without major redesign, by upgrading of the CPU chip or adding extra memory chips. However, chips are not tolerant of noise[2] in their input. And consequently, computers are not tolerant of noisy input or internal faults.

Computers, and their networking software, are the components of the InterNet. Networking software and hardware conforms to simple and strict communication protocols. Consequently, it is easy to incorporate new computers into the InterNet. These protocols include methods for detecting and recovering from errors and noise. Consequently, the InterNet as a whole is a very robust system.

[2] I.e. incorrect bits. They are tolerant of minor fluctuations in voltage levels.

Neural networks have the four properties listed above. All neural networks map from an input vector to an output vector. They degrade gracefully as the noise in their input vector increases. There are neural network algorithms for many types of problems: associative memories, clustering inputs, feature detection, and learning arbitrary mappings. These algorithms also vary in their performance characteristics: some learn very quickly, slower learning algorithms may store more information per weight than quicker algorithms, some algorithms can learn a wider range of tasks than others.

Consequently, we should be able to design large systems of neural networks in such a way that these systems are robust and flexible.

4 Characterisation of Algorithms

In this section, those characteristics which differentiate the interfaces to neural network algorithms are listed. These characteristics are used in selecting algorithms for use in larger systems.

Mapping There are three major types of mapping implemented by neural networks: feature detection and clustering, associative memories, and arbitrary mappings. Some algorithms which implement arbitrary mappings can only learn mappings which are linearly separable, others can learn any mapping.

Vector elements All neural networks map from an input vector to an output vector. Both of these vectors can have either binary or real elements. Most algorithms accept any form of input vector, though a few are restricted to binary inputs. There is considerable variation in the form of the output vector. Most algorithms generate either a binary vector or a bounded real vector[3]. However, some algorithms generate vectors of unbounded reals.

Training requirements Associative memory and feature detecting networks typically require only a set of example input vectors. This is known as unsupervised learning. Networks which learn arbitrary mappings require a set of example input vectors, and some indication of the mapping they should approximate. Usually this is a set of desired output vectors, one for each vector in the example input set. This is called supervised learning. However, some algorithms only require a scalar measure of the accuracy of their output vectors. This is called reinforcement learning. Algorithms for all types of mapping vary in the number of example input vectors they will need to learn an acceptable mapping.

Speed of recall In most algorithms, the activation of each node is computed only once during recall. Within these fast recall algorithms, there are some which use only simple addition and multiplication operations, and are extremely fast. Other algorithms use the sigmoidal squashing function, which is comparatively slow. The slowest recall algorithms are the relaxation algorithms, which use an iterative recall procedure.

Training speed Most neural networks use iterative training procedures, and train quite slowly. However, some networks use one–shot training procedures, and train quickly.

Deterministic Most neural networks are deterministic: for a given input vector and given set of weights, they will always produce the same output vector. However some are non–deterministic. Non–deterministic networks include the Boltzman machine, Hopfield nets with random node update, and A_{R-P}.

Stability of learnt mapping During training, the mapping implemented by most algorithms changes, even for example inputs for which a correct response has previously been learnt. Thus, most neural networks are trained off–line. However, some algorithms are designed so that the mapping of existing example input inputs is not altered by training on new example inputs. These algorithms can be trained on–line.

5 Example Characterisations

Unless otherwise noted, the algorithms characterised here have deterministic recall, and are unstable during training. The descriptions of training and recall speed given in this section are qualitative, and are intended as a basis for comparing the different algorithms.

[3] whose elements are in either the range $0 \ldots 1$ or $-1 \ldots +1$.

Correlation matrix algorithms [Aus87] map from a real vector to either a binary vector or real vector ; can only learn linearly separable tasks and are not guaranteed to learn any given linearly separable task ; recall is very fast ; training is one–shot and very fast ; supervised learning.
Perceptron [MP69] maps from real vector to binary vector ; guaranteed to learn any linearly separable task ; recall is very fast ; training is iterative but fairly quick ; supervised learning.
Single layer back propagation maps from a real vector to a bounded real vector ; learns arbitrary linearly separable tasks ; recall is fast ; training is iterative and slow ; supervised learning. This algorithm can propagate an error vector from its outputs to its inputs. It is this this ability that makes the multi–layer back propagation algorithm possible.
Multi–layer back propagation [RHW86] maps from a real vector to a bounded real vector ; learns arbitrary tasks ; recall is fast ; training is iterative and very slow ; supervised learning. As a multi–layer network, the number of hidden nodes must be chosen. This is usually done manually, but algorithms exist which do this automatically [Ash89] [FL90].
A_{R-P} [Bar85] maps from a real vector to a binary vector ; learns arbitrary tasks ; recall is fast but may be non–deterministic ; training is iterative and very slow ; reinforcement learning ; the number of hidden nodes and topology must be determined manually.
Boltzman machine [AH85] maps from a binary vector to a binary vector ; learns arbitrary tasks ; recall is iterative, slow and non–deterministic ; training is iterative and very slow ; supervised learning.
Kohonen [Koh90] maps from a real vector ; learns important features of its example inputs ; recall is fast ; training is iterative and slow ; unsupervised learning. There are many ways of representing the output of a Kohonen net as a vector.
Hopfield networks [Hop82] map from a binary vector to a binary vector ; are auto–associative memories ; recall is iterative but fairly fast ; recall is non–deterministic ; training is one–shot and very fast ; unsupervised learning.

6 Design of Neural Network Systems

This section discusses issues in designing systems whose components are neural networks, and which must learn arbitrary mappings from input to output.

As in feedforward neural networks, in systems of neural networks activation will flow from the input to the output. Those networks near the input can be termed the *early* networks, and those nearer the output can be termed the *late* networks.

A feature of the neural network paradigm of computation is that a neural network is comprised of many simple homogenous components. A user training a neural network to solve a task has to put little effort into hand–crafting a topology, or into the individual selection of components.

This feature does not carry over to systems comprised of neural networks. The system designer has to examine the desired function of each part of the system, and select an algorithm which is suited to that function.

The designer also has to consider the representation of data between networks. In particular, the designer needs to consider the size of intermediate vectors, whether an intermediate vector is to be invariant with respect to transformations of the input, and the degree to which the mapping required of a network is linearly separable.

The representation of numeric data in vectors is also important. It is important that the activation of a node increases monotonically[4] with the strength of the feature it represents. Thus, coding numeric values using binary coding[5] is a poor representation. If the vector between two layers is binary, it is best to represent numeric values with thermometer codes[6].

[4] (or decreases monotonically)
[5] I.e. 000, 001, 010, . . . , 111
[6] I.e. 00000000, 00000001, 00000011, . . . , 11111111.

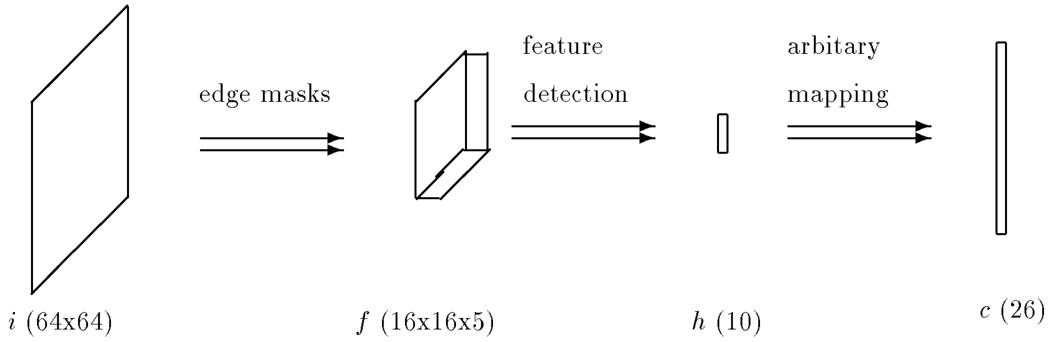

Figure 1: Architecture of example design

If training data is available for an intermediate vector, as well as for the output vector, the intermediate data should be utilised. For example, a system may be required to recognise words independent of the speaker. If the training inputs can be broken up into useful categories, such as male vs female, the system should have two parts. The first network(s) should compute the sex of the speaker from the system's inputs. The remaining network(s) should recognise the spoken word, using both the system's inputs and the computed sex of the speaker. Yu and Simmons [YS90] propose a similar scheme.

The early networks in a system are usually data driven. They take the input, and transform it into a different representation in which significant features of the input are represented on separate axes. Even if information about the desired mapping is available, it might not be used to train the early networks.

If the system is to be invariant with respect to well defined transformations of its input, such as translation and rotation of images, this invariance is best incorporated in an early layer. Many algorithms can have this kind of invariance hardwired into them. Reid et al. [RSO89] and Giles and Maxwell [GM87] use polynomials with restrictions on the coefficients which are invariant with respect to translations, rotations and magnifications of their inputs.

For many task domains, such as image processing, the early layers have much larger input vectors, and consequently have the bulk of the compuational load. For these tasks, it is more important to select efficient algorithms for the early networks than for the late networks.

The late networks are driven by the desired mapping. The form of training data will affect the kind of network which can be used here. In particular, the range of possible algorithms is much restricted if desired outputs are not available for the training inputs.

The earliest networks in the system must be trained first. In Figure 1, the mapping from i to f is hard coded, and does not need training. The mapping feature detection mapping from f to h must be trained before the arbitary mapping from h to c, because the feature detection mapping needs only example input vectors, but the arbitary mapping needs example input and output vector pairs. There is an exception to this rule: where the algorithm used in a latter layer can propagate training information to the previous layer, the two layers can be trained simultaneously.

7 A Design Example

In this section, the principles described above are demonstrated in the design of a system whose input, i, is a 64x64 binary image of a letter as its input and outputs the corresponding letter in a 26–element vector, c. We assume we have a large set of image to class training examples.

The system has three layers, as show in Figure 1. The first layer is hard–wired, reduces the size of the vector and extracts generally useful image processing features. The second layer is a trained feature detector. The third layer learns to map the image features detected by the second layer onto the desired outputs.

The first layer maps the input image i onto a 16x16x4 image, f. A window around each fourth pixel in i is mapped to a four element vector in f. The four elements are the result of convolving with ridge detecting masks[7] at four orientations.

The second layer convolves the image f to produce a feature vector h, with about ten elements. The exact size of h is best determined by trial and error. The mapping from h to c should be linearly separable.

The third layer must learn the mapping from the feature vector h to the class vector c, so it must be a network capable of learning arbitrary mappings. This layer is smaller in size than the previous layers, so the speed of the algorithm used here is not important.

We examine several possible configurations of the network.

Multilayer back propagation Use a single layer of back propagation in both the second and third layers. The second layer can be trained using the back propagated information from the third layer. The greatest disadvantage of this system is the very slow learning of multilayer backpropagation networks.

Kohonen and perceptron Use a Kohonen net in the second layer, and a perceptron in the third layer. This net is reasonably quick to train and during recall.

Kohonen and correlation matrix This is similar to the above configuration, but not as good. The correlation matrix algorithm is not guaranteed to learn any linearly separable mapping. It trains more quickly than the perceptron, but this is not an important consideration as the training time is dominated by the second layer.

A mixed network If speed and simplicity during recall, at the expense of training complexity, are desired then a system might be built in stages, as follows.

First, train two layers of back propagation as the second and third layer.

Next, replace the back propagation second layer with a correlation matrix network. Using the third layer to back propagate a desired output, train the new second layer.

Finally, replace the back propagation third layer with a correlation matrix network, and train.

The system now consists of a hard wired layer, and two correlation matrix layers, all of which are very simple and very fast.

This example design has illustrated the major points of this paper. A multi–layer system has been designed. The function of each layer and the information represented in the intermediate vectors was decided. This design could be implemented by several configurations of networks, each configuration having its own training, recall and complexity trade–offs. It is also possible to alter the configuration of a network during training.

The usefulness of propagating training information backwards through a network has been demonstrated.

8 Research Directions

Many algorithms have continuous valued outputs. The conversion of these continuous vectors for input to networks which have binary input vectors should be investigated.

Where a trained network is to be removed from a system and replaced with another network, it would be useful to be able to use the trained weights of the old network as an initial set of weights for the training of the new network. This area should be researched.

The analysis given in this paper ignores networks which have a concept of state, such as recurrent back propagation networks. The approach should be extended to include such networks.

Few algorithms provide training information for earlier networks. The back propagation algorithm of Rumelhart et al. [RHW86] is one such algorithm. Other algorithms, including correlation matrix and even thresholding networks, could provide some training information for earlier networks.

[7] Convolving with a mask is not a neural network algorithm. However, it fits neatly into the interface described in § 4.

9 Conclusions

The desirable properties of components for building large systems which are flexible and robust have been described. The components should communicate simply, be noise tolerant and interchangeable. Neural networks have these properties.

A characterisation of neural networks suitable for use in designing systems of neural networks has been given. To demonstrate this characterisation, it has been applied to some well known algorithms.

Some issues involved in designing systems of neural networks have been described, and an example design given. The example design demonstrates replacing one algorithm in a working system with another algorithm of similar function but different performance characteristics.

Most neural network research has been devoted to the development or application of a particular algorithm. We believe the interactions of neural networks and their use in large systems should be explored.

10 Acknowledgements

This work is supported by the SERC grant GR/F36330.

References

[AH85] D.H. Ackley and G.E. Hinton. A learning algorithm for boltzmann machines. *Cognitive Science*, 9:147–169, 1985.

[Ash89] T. Ash. Dynamic node creation in backpropagation networks. Technical Report ICS Report 8901, University of California at San Diego, February 1989.

[Aus87] J. Austin. *The Design and Application of Associative Memories for Scene Analysis*. PhD thesis, Department of Electrical Engineering, Brunel University, Uxbridge, London, UK, 1987.

[Bar85] A.G. Barto. Learning by statistical cooperation of self-interested neuron-like computing elements. *Journal of Biological Cybernetics*, 4:229–56, 1985.

[FL90] S.E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. Technical Report CMU-CS-90-100, Carnegie Mellon University, February 1990.

[GM87] C.L. Giles and T. Maxwell. Learning, invariance and generalization in high-order neural networks. *Applied Optics*, 26(3):4972–8, 1987.

[Hop82] J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79:2554–2558, April 1982.

[Jac90] R. A. Jacobs. *Task Decomposition through Competition in a Modular Connectionist Architecture*. PhD thesis, Department of Computing and Information Science, University of Massachussets, Amherst, MA 01003, 1990. Also published as COINS technical report 90-44.

[Jac91] R. A. Jacobs. Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87, 1991.

[Koh90] T. Kohonen. The self–organising map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[Kup90] M. Kuperstein. Infant neural controller for adaptive sensory–motor coordination. *Neural Networks*, 4(2):131–145, 1990.

[MP69] M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969.

[Mun87] P. Munro. A dual back–propagation scheme for scalar reward learning. In *Proceedings of the 9th Conference of the Cognitive Science Society*, pages 165–176, 1987.

[NW89] D. Nguyen and B. Widrow. The truck backer–upper: An example of self–learning in neural networks. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, pages 357–363, June 1989.

[Pom91] D. Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, 3(1):88–97, 1991.

[Pom92] D. Pomerleau. *Neural Network Perception for Mobile Robot Guidance*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.

[RHW86] D. Rumelhart, G. Hinton, and R. Williams. Learning representations by back propagating errors. *Nature*, 323:533–6, 1986.

[RSO89] M.B. Reid, L. Spirkovska, and E. Ochoa. Simultaneous position, scale and rotation invariant pattern classification using 3rd order neural networks. *International Journal of Neural Networks: Research and Applications*, 1(3):154–9, 1989.

[YS90] Y-H. Yu and R.F. Simmons. Extra output biased learning. Technical Report AI90-128, Artificial Intelligence Laboratory, University of Texas at Austin, 90.