Data-Defined Problems and Multiversion Neural-Net Systems

Derek Partridge & William B Yates

Department of Computer Science
University of Exeter, Exeter EX4 4PT, UK
derek@dcs.exeter.ac.uk

ABSTRACT

Data-defined problems are not restricted to any particular problem domain, and are common. Data-defined problems, as the name suggests, are defined by a set of input-output mappings, and for the problems of particular interest the full details of how the inputs are related to the outputs are unknown. Such problems present the traditional programmer, whether using AI techniques or not, with a difficult task. This is because programming requires a prerequisite understanding of the mechanisms relating input and output such that an algorithmic solution can be devised. Automatic induction techniques demand no such prerequisite information. Neural computing is one such induction technique. And, moreover, it is one that can outperform more traditional AI induction techniques, such as IF-THEN rule systems. Neural computing can do better because it can follow a well-defined path to an optimal result, and because multiple, alternative versions can be cheaply generated to permit exploitation of certain properties of the resultant version set – in particular, 'diversity'. We define a valuable extension to previous definitions of this quantity, and exploit it to produce significant system performance enhancements. In addition, because our neural computing is an approximating technique, it is also amenable to exploitation of diversity through averaging multiple versions. An example of letter recognition is used to illustrate these ideas.

KEYWORDS

data mining, neural computing, software diversity, multiversion software, voting, averaging

1. INTRODUCTION

Computational solutions to problems typically rely on the existence of an abstract specification of the problem to be tackled. But where do such specifications come from? With the important, but unrepresentative, exception of abstract technical problems (such as greatest common divisor, or prime numbers), the specification is generated by someone (perhaps system analyst) from the real-world manifestation of the problem.

This specification may be a 'good' one (i.e. accurate and useful), or a 'bad' one. It is, however, both a fiction and an approximation of reality. But it can be an invaluable stepping stone from the problem to an effective computational solution. In traditional software engineering, the specification is expected to be complete and accurate, whereas in AI it is more acceptable for the specification to be known to be both incomplete and in some ways no more than a rough guide (which is developed and refined using evolutionary programming techniques, see Partridge, 1994). But in either case, there is a prerequisite need for *some* reasonable understanding of the input-output relationships of the problem in order to guide the programming task.

Many problems, however, are manifest as little more than sets of input-output data. They exist in systems of high complexity where our knowledge of the underlying mechanisms is both crude and fragmentary. Some examples of such data-defined problems are: human face recognition; signature recognition; prediction of periodic fluctuations in water consumption or electricity demand for a city; optimal control of complex chemical processes and manufacturing plants; adjustment of treatment dosage on the basis of bodily response to last dosage, etc. In all of these cases, it is far easier to collect examples of the problem data (both good and bad examples) than it is to determine in more than a very rough and fragmentary manner precisely how the output depends on the input.

Effective computational solutions to such problems must come through data-driven techniques such as automatic induction. An induction algorithm constructs a general solution (such as a decision tree, a rule set, or a neural network) from a set of examples. To be useful the induction techniques must interpolate between and extrapolate beyond the given examples.

Supervised learning in neural computing is one such induction technique, and one that appears to be more powerful than the more traditional AI induction techniques, such as rule-based classifier systems (e.g. Holland *et al.*, 1986).

2. WHY NEURAL COMPUTING IS DIFFERENT

There are many different manifestations of neural computing. In this paper we are only concerned with supervised learning in parallel distributed neural-net systems. The classic example (but by no means the only one) is the training of multilayer perceptrons (MLPs) using the backpropagation (BP) algorithm (Rumelhart and McClelland, 1986).

A conventional program is a symbolic structure with special properties (apart from being machine executable). Programs exhibit a compositionality of both syntax and semantics such that the meaning of the whole program can be composed systematically from the meanings of the individual components. And, moreover, the components themselves relate directly to the elements of the conceptual solution to the problem.

Thus if we program a system to recognize handwritten letters, then components of the program will map onto elements of our conceptual solution. For example, if we believe that the proportion of vertical to horizontal line is an important determinant, then the program will contain components that prescribe precisely how to compute the proportion of vertical line in any character image, and in what way it is used in the recognition process. Not only will components of the program relate to our understanding of how to solve the problem, but the dynamic processes prescribed in the program will also similarly reflect elements of the same understanding. The result of this is that we can rationalize the operation of the program in terms of our understanding of how the problem should be solved.

A trained MLP offers none of these useful correspondences. There is no known way to directly relate components of a trained MLP to elements (either static or dynamic) of our conceptualization of the problem that the trained network solves. This absence of useful (some might say, essential) relationships is not thought to be beneficial, and many researchers are endeavouring to elicit some valuable correspondences between problems and MLP implementations. However, for data-defined problems, when we don't even possess a substantial conceptualization of the problem anyway, such efforts might seem rather pointless – unless they facilitate the development of a better conceptualization of the problem. But if we still believe that computerization of such problems can be useful, then neural computing provides an appropriate technology.

3. NEURAL COMPUTING AS A SOFTWARE TECHNOLOGY

Inductive programming using MLPs is radically different from conventional programming (see Partridge, 1995). Because it is a programming novelty, new possibilities for software engineering emerge. Ready applicability to data-defined problems (as mentioned above) is just one such possibility; several others are introduced below

3.1. MLP Implementations

In order to generate an MLP implementation of a problem we require a representative sample of the input-output data, a ***training set***, and an initial network that is capable of learning this set. The main architectural decision in regard to the initial network is how many hidden units (i.e. units connecting input units to outputs units) to use (and perhaps how many layers of hidden units, but we restrict our system to just one layer of hidden units). This decision, which only needs to be approximately correct (especially in the context of the multiversion strategy we employ), is usually reached by the application loose guidelines together with some preliminary training experiments (which can also serve to investigate and to set the few parameters required by the BP algorithm).

Other architectural decisions are not pressing, although there is increasing recognition that considerable benefits can accrue if problem-specific knowledge can be used to pre-structure the network. The numbers of input and output units are largely determined by the complexity of the input and output of the problem being implemented. Finally, such initial networks are usually totally connected (i.e. each input unit has a link to every hidden unit, and each hidden unit links to every output unit), and the initial link weights are randomly assigned (usually within a small range centred on zero).

The network is then trained using the BP algorithm (which iteratively adjusts all the link weight values) to correctly compute the training set of examples. This is achieved when some predefined convergence criterion is attained – e.g. 95% of training-set data each learned to within 10% of the ideal result. The link weights are then fixed, and the trained MLP is a deterministic implementation of some generalization of the training-set data.

The training process does not always converge, and even when it does, it does not always converge on a significant generalization of the training set. In either case, it is usual to re-examine the parameter settings and network

architecture, and to retrain under some modification. It may be that all that needs altering is the random initialisation of the link weights. The nature of the trained network (in terms of correctness of performance on a test data set) is known to be highly sensitive to the initial conditions of training. It is thus usual to train more than one version, and to then select the best based on test-set results – i.e. it is accepted that single-net training is not a robust technique. We exploit this feature of the technology and turn it into an advantage rather than accept it as a drawback.

3.2. Exploiting the Economics of Automatic Approximation

Because it is an algorithmically determined, automatic process, training MLPs is relatively cheap and quick. It can involve considerable computer time, especially when the massive parallelism must be simulated sequentially on a conventional computer, but very little programmer time. This radical change in the nature of the 'programming' task produces a similarly profound change in the economics of programming.

Because all practical programming technologies are error prone, it has long been accepted that by programming the same problem in several different ways a set of alternative versions can be obtained, and that this set can be used as a *multiversion system* to deliver a more accurate performance than any of the component versions. In traditional software engineering, which has toyed with this idea but generally shied away from it as too expensive for day-to-day programming tasks, the multiversion system is conceived of in conjunction with a voting strategy. Thus all component versions may be surveyed on their individual decisions, and a majority vote taken to determine the overall result. The efficacy of such a voting strategy relies on *diversity* within the version set – i.e. the individual versions have a minimum of failures in common, if one version is incorrect then the others are likely to be correct.

Inability to justify the large increase in costs that multiversion software systems demand, is reinforced by the fact that empirical studies show that the usefully high levels of diversity are not forthcoming. Even when the multiple systems are developed quite independently of each other, statistically they do not fail independently – i.e. if one version is incorrect on a given input then the others are also likely to be incorrect. So, not only is it prohibitively expensive to manually program multiple versions of a problem, it is also impossible to control the various programming exercises such that

the resultant versions are diverse enough to deliver a reasonable reliability increase that might offset the very substantial extra costs.

But, in our current neural computing context, the development of multiple versions is cheap, *and* the error characteristics of the trained versions are determined solely by the initial conditions for training – the nature of the training set, of the network architecture, and of the few parameter settings. This latter feature means that we have the opportunity to *systematically engineer* high diversity levels into a multiversion neural-net system. We have developed diversity measures (Krzanowski and Partridge, 1995), explored the sensitivity of important features of the initial conditions to version-set diversity (Partridge, 1996), and developed systematic strategies for generating highly diverse version sets (Partridge and Yates, 1996).

The final point that needs to be made about the use of MLPs as a software engineering technology concerns that fact that they are continuous approximators rather than discrete computational devices like conventional programs. This feature suggests that they will not be good for, say, precise numerical computation where anything other than the precisely correct answer is totally wrong. But on the positive side, it opens a new option for multiversion decision strategies, in addition to voting we can use averaging.

Previous studies of multiversion techniques have been based on the distinction between success and failure, between correct and incorrect computations on test inputs. We shall expand on this basis in two ways: first, we differentiate among incorrect computations to exploit diversity among wrong answers, and second, we explore a strategy of shading the hard distinction between correct and incorrect into a continuum of approximate correctness.

In order to provide a source of concrete illustrations for the theoretical framework to be developed, a specific example problem will be introduced.

4. LETTER RECOGNITION: AN EXAMPLE APPLICATION

A difficult data-defined problem is that of letter recognition as presented by Frey and Slate (1991). The data that defines this problem consists of 20,000 unique letter images composed of the letters A to Z from 20 different fonts (publically available aha@ics.uci.edu). Each was distorted both horizontally and vertically but still remained "recognizable to humans".

Sixteen features were then defined to capture specific characteristics of the letter images, characteristics concerning the 'strokes' that constitute a letter and how they are interrelated. The value of each feature in each image was linearly scaled to an integer in the range 0 to 15 (inclusive). Each of the original letter images was thus transformed into a list of 16 such integer values. It was a file of 20,000 such integer vectors, each associated with its correct letter, that was the data that defined this problem.

Notice that this data-defined problem would pose enormous problems for the wholly conventional programmer. Before development of an algorithm could begin, a comprehensive and detailed understanding of the actual relationships between the 16 features and the 26 letters would, somehow, need to be generated. It is not at all clear how such an understanding, which is a necessary prerequisite of the programming task, could be obtained at all. This is clearly a problem better suited to an inductive implementation technology such as the field of AI offers.

Frey and Slate use the first 16,000 image vectors to develop a rule-classifier system which they then tested on the remaining 4,000 image vectors. They report a comprehensive study of a variety of options for controlling the development of their systems. They obtain a few combinations of options which result in systems that exhibit greater than 80% correctness on the test set – actually, 80.8%, 81.6% and 82.7% are the three best. In an earlier study (Partridge and Yates, 1995), we showed that a simple neural computing approach to this problem delivered a better performance – just over 90%, and up to 86% using only 12% of the 16,000 training resources – using less resources, and with no parameter tuning. In this study, we define a new measure of diversity and use it to systematically engineer an optimal neural-net implementation. The new measure is an extension of the previous ones, an extension that is designed to measure the diversity among incorrect results.

5. DIVERSITY AMONG DISTINCT ERRORS

Previous studies that have defined measures of diversity (e.g. Littlewood and Miller, 1986; Partridge, 1996; Krzanowski and Partridge, 1995) have all collapsed the variety of possible error to a single error category. The resultant diversity measures are then based upon the property of whether alternative versions are likely to be in error coincidently or not, and this is

used as a crucial determinant of multiversion system potential. However, if the problem is one that admits variety among the erroneous results (i.e. more than 1 distinct error category), then further scope for multiversion diversity (and subsequent exploitation) appears. The previous diversity measures can be extended to include this 'distinct error' diversity as well as the 'coincident failure' diversity, *CFD* of Krzanowski and Partridge (1995).

5.1. Distinct-error Diversity Definition

Assume that we have a set of N versions, each trained to implement a categorization problem in which there are c distinct categories, and that this set is subjected to M test cases. In the situation in which all versions are not identically perfect (otherwise CFD=0 by definition), coincident-failure diversity has been defined as:

$$CFD = \sum_{n=1}^{N} \frac{(N-n)}{(N-1)} f_n$$

where $f_n = \dfrac{\textit{the number of tests that fail on exactly n versions}}{\textit{total number of tests that fail on at least one version}}$. So f_n is the probability that a test will fail on exactly n versions. Thus the less coincident failures in the version set, the higher the values of f_n for small n, and hence the higher the (coincident-failure) diversity of the set.

The CFD measure ranges from 0 when f_N = 1.0 (i.e. all failures are common to all versions, hence no diversity) to 1.0 when f_1 = 1.0 (i.e. all versions fail uniquely, no coincident failures, hence maximum diversity).

But in a situatiton of more than 1 distinct possible errors, the CFD measure, which does not distinguish whether n failures are all distinct or all identical, represents a lower bound on the diversity of the version set.

In order to measure the diversity generated within the (c–1) distinct failure categories associated with any output (it is (c–1) because one category will be the correct output), we define a new probability, t_n, the probability that exactly n versions will fail identically on a test case.

$$t_n = \frac{\textit{the number of times that exactly n versions fail identically on a test}}{\textit{the total number of times that at least one version fails distinctly}}$$

This probability can then be used to provide a measure of distinct-failure diversity, *'DFD* (again provided not all versions are perfect, otherwise *'DFD* = 0).

$$DFD = \sum_{n=1}^{N} \frac{(N-n)}{(N-1)} t_n$$

which has a minimum value of 0.0 when t_N = 1.0 (i.e. all versions fail identically), and a maximum value of 1.0 when t_1 = 1.0 (i.e. all failures are uniquely distinct).

However, notice that if $c \geq N$, then all failures may be distinct but all versions may fail on all tests! To rectify this unsatisfactory situation, the overall diversity must include both *CFD* and *'DFD*. One possible combination is simply the geometric mean. Overall diversity, *OD* = $\sqrt{CFD \times 'DFD}$.

This measure still ranges from 0.0 to 1.0, and moreover in the awkward case pointed out above, i.e. all versions fail distinctly on all tests, *'DFD*=1.0 but *CFD*=0.0 (because f_N = 1.0) and so *OD*=0.0 as it should. Notice that in situations of no distinct failures $f_n = t_n$, and so *'DFD* = *CFD*. Hence *OD* = $\sqrt{CFD^2}$ = CFD which is in accord with the earlier observation that coincident-failure diversity is a minimum diversity that can be increased by consideration of the distribution of system failures among a set of distinct errors.

6. EMPIRICAL STUDY

In this section we present the results of our attempts to engineer and exploit the maximum diversity within sets of networks trained on the letter recognition problem which offers the potential of 25 distinct failures to exploit. We employ a technique of 'over produce and choose' (Yates and Partridge, 1996) in which we generate a 'space' of trained versions and then choose a maximally diverse subset as the final multiversion system.

As a result of training networks, both MLPs and Radial Basis Function (RBF, see Partridge and Yates, 1996, for details) nets, under a variety of initial conditions, we obtained a 'space' of 132 (68 MLPs and 65 RBFs) alternative trained versions for the OCR problem – a *version space*.

Using a procedure designed to select a subset of versions from this space under the constraint that some version-set measure should be maximized, three version sets, each containing nine networks, were selected. This selection procedure, known as the *pick heuristic* (Partridge and Yates, 1996), has been extended to favour selection of diverse versions that also succeed on 'difficult' inputs. "The difficulty in processing input x" (the $\theta(x)$ term in Littlewood and Miller's (1986) model, p. 1597) is derived from the number of versions, in the complete version space, that fail to compute a correct result for each input. Thus modified, the *pick heuristic* was used to select from the complete version space in accordance with the maximization of three different measures: *OD*, *CFD*, and *DFD*. In this way three multiversion systems, $pick_{OD}$, $pick_{CFD}$ and $pick_{DFD}$ respectively, were available for assessment using the final 4000 data items in the original database. A 'success' was recorded when a single maximum output category was that of the target, otherwise a 'failure' (for that test on that version) was recorded. The initial results are given in Table 1. The $pick_{DFD}$ system was composed of 4 RBF nets and 5 MLPs, $pick_{OD}$ contained 2 RBF nets, and $pick_{CFD}$ contained only 1 RBF which was common to all three systems. The three systems had 6 nets in common.

Table 1
Test results on the 9-version systems selected

	OD	*CFD*	*DFD*	*aver*	*max*	*min*	*maj. vote*
$pick_{CFD}$	0.890	0.836	0.948	79.81%	88.62%	33.77%	85.87%
	(0.875)	(0.814)	(0.942)				
$pick_{DFD}$	0.842	0.730	0.972	69.92%	88.62%	33.77%	81.57%
	(0.829)	(0.709)	(0.969)				
$pick_{OD}$	0.887	0.815	0.966	66.95%	88.62%	36.68%	84.47%
	(0.874)	(0.794)	(0.961)				

Each diversity measure has two values recorded: diversity with respect to the 'picking' set of 16,000 items, and in parentheses diversity with respect to the test set of 4,000 items. The column headed "aver" gives the average percent correct over the nine versions, "max" indicates the percent correct of the best network (which is the same network in all systems), "min" is the worst network (an RBF in all systems, and the same one in $pick_{CFD}$ and $pick_{DFD}$), and "maj. vote" gives the system performance under a simple

majority-vote strategy that considers all incorrect answers as failures, so it is the majority of successes over failures.

The results reveal high levels of both types of diversity within the multiversion systems, and moreover, the highest *DFD* value is found in system $pick_{DFD}$ and the highest *CFD* value is exhibited by system $pick_{CFD}$ which is as expected. The highest *OD* value is not exhibited by the $pick_{OD}$ system, and this is possible because the 'pick' procedure is a heuristic one which does not guarantee an optimum result. Notice that although the diversity values generated during the 'pick' process are always higher than those exhibited during testing, the *DFD* values are the least affected.

The success/failure majority-vote performance is 4% to 7% worse than the performance of the single best network in each system. This is despite the fact that the diversity values are high. However, these majority-vote performances are 6% to 17% better than the system averages. The conclusion is that the diversity potential of the systems (the sole basis on which they were selected) is being undermined by the low performance level of some of the component versions. It should be noted, however, that single-net performances can be highly dependent on specific test sets while multiversion system performances tend to be more robust in this respect.

Exploitation of distinct-failure diversity requires that detailed results, rather than just success or failure, are recorded and fed into the final decision strategy. Three different ways of doing this were explored:

averaging: The raw values computed in each output category was summed over all eight versions, and the output category with the maximum value was selected as the system result.

thresholding: The raw values for each version were thresholded at 0.5, to give the subset of categories that each version 'voted for.' These category votes (from 0 to 26 per version per test input) were surveyed over all nine versions, and the category with the most votes was selected as the system result.

winner-takes-all: The maximum value of the raw results for each version was used as that version's vote. These category votes (exactly one per version per test input) were surveyed over all nine versions, and the category with the most votes was selected as the system result.

All cases of no single maximum were treated as test failures.

The results obtained by exploiting distinct-failure diversity in these three ways are given in Table 2 using the same 4,000-item test set as previously.

Table 2
Test results from exploitation of distinct-failure diversity

	averaging	thresholding	winner-takes-all
$pick_{CFD}$	92.95%	92.00%	91.10%
$pick_{DFD}$	93.47%	89.57%	89.42%
$pick_{OD}$	92.85%	91.12%	90.85%

The first point to notice is that all results are an improvement on any in Table 1. They range from 12% improvement over simple majority vote for $pick_{DFD}$, to less than 1% improvement of the winner-takes-all strategy over the best individual network in the same system. Secondly, the averaging strategy is consistently superior to thresholding which tends to be better than winner-takes-all.

In the above strategies for exploiting distinct-failure diversity, all component versions are treated equally as contributors to the final system outcome. It seems reasonable to expect that some versions are 'better' overall than others and therefore should be given more 'say' in the overall result. In order to test this possibility, a *decision net* (DN) was constructed by means of a weighted link from each component net to the final decision strategy. Each version's contribution to the overall system outcome is then proportional to its weight. Using the original 16,000 training items (it would have been preferable to use a totally new set, but no such set was available and the 4,000-item set was strictly reserved for final testing), the nine link-weights were adjusted using Widrow-Hoff learning to achieve an optimal performance under the best of the three decision strategies – averaging. The resultant three decision-net systems were then tested using the averaging decision strategy. The results are given in Table 3.

Table 3
Test results from learned priorities

$DN_{pick_{CFD}}$	94.62%
$DN_{pick_{DFD}}$	94.07%
$DN_{pick_{OD}}$	94.55%

These final results illustrate that there are also non-trivial improvements to be made by properly weighting the contributions from each of the component networks in a multiversion system. An approximately 1% additional improvement is also a further 15% reduction in the residual error in the systems.

7. CONCLUSIONS

The results serve to confirm the previous conclusion that neural computing is a more effective technology than traditional AI techniques, such as rule induction, for application to data-defined categorization problems exemplified by the letter recognition task. The current results clearly demonstrate how much more effective this technology can be when the full diversity potential is generated and exploited.

Notice, however, that 'standard' BP is all that has been used to generate these results. There are many 'improved' versions of BP, and use of one of these may well produce a further significant increase in overall performance. The measure of distinct-failure diversity was defined (*DFD*) and appears to be a valuable extension of the previously defined coincident-failure diversity measure (*CFD*). High levels of this diversity were shown to be obtainable. More significantly, it was demonstrated that proper exploitation of this diversity 'extension' can result in significant system performance enhancements (approximately 10% improvement eliminating two thirds of the residual error). This diversity measure was effectively exploitable using a majority-in-agreement voting strategy instead of the more usual success/failure majority-vote strategy.

The results indicate that the most effective exploitation of distinct-failure diversity is to delay thresholding decisions, by averaging individual version

performances, thus capitalizing on the approximating nature of this particular inductive technology. Furthermore, this averaging strategy yields a further non-trivial performance improvement as a result of optimizing the relative contribution of each version to the overall system outcome.

8. REFERENCES

1. Frey, P.W. & Slate, D.J. 1991. Letter recognition using Holland-style adaptive classifiers, *Machine Learning*, **6**, 161-182.
2. Holland, J.H., Holyoak, K.J., Nisbett, R.E. & Thagard, P.R. 1986. *Induction: Processes of Inference and Discovery*, Cambridge, MA: MIT Press.
3. Krzanowski, W.J. & Partridge, D. 1995. Software diversity: practical statistics for its measurement and exploitation, Res. Rep. 324, Dept. of Computer Science, University of Exeter (submitted to *Information & Software Technology*).
4. Littlewood, B. & Miller, D.R. 1986. A conceptual model of coincident failure in multiversion software engineering, *IEEE Trans. on Software Engineering*, **15**, 1596-1614.
5. Partridge, D. 1994. *Engineering AI Software*, Intellect Books: Oxford, UK or Ablex Pub. Corp., NJ, USA.
6. Partridge, D. 1995. On the difficulty of really considering a radical novelty, *Minds and Machines*, **5**, 391-410.
7. Partridge, D. 1996. Network generalization differences quantified, *Neural Networks*, 9, 263-271.
8. Partridge, D. & Yates, W.B. 1995. Letter recognition using neural networks: a comparative study, Res. Rep. 334, Dept. of Computer Science, University of Exeter.
9. Partridge, D. & Yates, W.B. 1996. Engineering multiversion neural-net systems, *Neural Computation*, **8**, 869-893.
10. Yates, W.B. & Partridge, D. 1996. Use of methodological diversity to improve neural network generalisation, *Neural Computing & Applications*, **4**, 2, 114-128.
11. Rumelhart, D.E. & McClelland, J.L. 1986. *Parallel Distributed Processing*, MIT Press: Cambridge, MA.