Categorization of Programs Using Neural Networks

Franz J. Kurfess Lonnie R. Welch

Computer and Information Science Department
New Jersey Institute of Technology
Newark, NJ 07102

Abstract

This paper describes some experiments based on the use of neural networks for assistence in the quality assessment of programs, especially in connection with the reengineering of legacy systems. We use Kohonen networks, or self-organizing maps, for the categorization of programs: Programs with similar features are grouped together in atwo-dimensional neighbourhood, whereas dissimilar programs are located far apart. Backpropagation networks are used for generalization purposes: Based on a set of example programs whose relevant aspects have already been assessed, we would like to obtain an extrapolation of these assessments to new programs. The basis for these investigation is an intermediate representation of programs in the form of various dependency graphs, capturing the essentials of the programs. Previously, a set of metrics has been developed to perform an assessment of programs on the basis of this intermediate representation. It is not always clear, however, which parameters of the intermediate representation are relevant for a particular metric. The categorization and generalization capabilities of neural networks are employed to improve or verify the selection of parameters, and might even initiate the development of additional metrics.

1 Introduction

The development of large software systems does not end with the installation of the executables on the target system: Apart from bug fixes, changes will have to be made to accomodate new functionalities, to integrate modifications in business practices, or to port the system to a new operating system or hardware platform. Whereas nowadays efforts are often made to consider these aspects in the development of new systems, existing systems – which might have been developed decades ago – are not necessarily amenable to major modifications. On the other hand, a complete redevelopment often is economically infeasible and unnecessary as long as the major parts of the system work satisfactorily. In a situation which requires major modifications, e.g. the transition from a mainframe to a client-server environment, a decision has to be made which parts of the software system should be kept, which ones should be modified, and which ones have to be completely rewritten. The basis for such a decision relies to some degree on strategic factors, e.g. the trustworthiness of a program, but should also consider aspects reflecting the quality of the system with respect to current software engineering practices. Whereas it is not really clear what exactly determines the quality of a program, a number of metrics have been developed which express certain properties of a program in a numerical way based on quantifiable features of the program. A well-known example of such a metric is the McCabe complexity [8]; others will be outlined below, and are described in more detail in other publications [13].

Organization of the Paper The rest of this paper describes our approach in more detail. Section 2 concentrates on the two types of neural networks used, and their properties relevant for our purposes. The third section gives a brief recollection of the intermediate representation which serves as the basis for the assessment of programs. Some assessment aspects have already been formalized as metrics, providing quantifiable statements about certain properties of programs; these metrics are outlined in Section 4. Sections 5 concentrates on the use of Kohonen maps for the categorization of programs, while Section 6 discusses the use of backpropagation nets for learning and generalization in connection with metrics. The final section provides a summary and conclusions.

0-8186-7355-9/96 $05.00

2 Neural Networks Used

In this paper, we investigate the usage of neural networks to assist with the quality assessment of software systems. We use two different types of networks: self-organizing feature maps (also known as Kohonen maps), and multilayer feedforward nets with the backpropagation learning algorithm. In this paper we will only resume the fundamental concepts of the neural networks used; for details, refer to any textbook on neural networks, e.g. [2, 4].

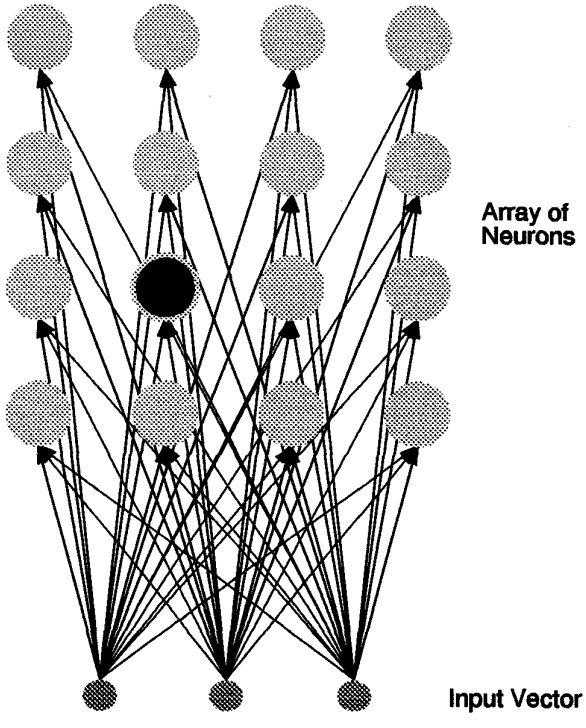

Figure 1: Kohonen Network

Kohonen Maps The Kohonen maps are used to categorize programs according to quality-relevant features; the goal is to group together programs which have similar properties.

The input for the network consists of a feature vector for each program out of the set of programs to be categorized. The feature vectors are derived from an intermediate representation of the programs, and utilize the same or similar features as used for the computation of the metrics. The output is a visual representation of the categorization result, where the items categorized (here: the vectors representing the programs) are arranged in a two-dimensional array; items with similar properties are in proximity to each other, whereas distinctly different items are further apart. Note that for this approach we do not need a set of example or test items which have been categorized before; the network only utilizes the set of items at hand.

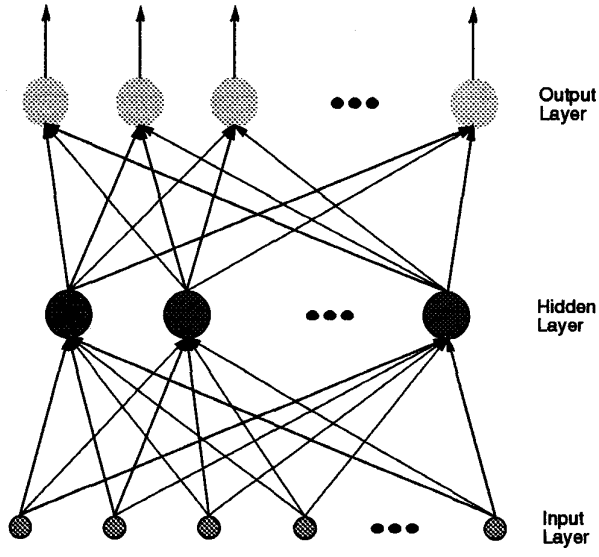

Figure 2: Backpropagation Network

Backpropagation Networks The second approach based on backpropagation networks, however, requires a set of examples together with their assessment. It uses these examples to derive a mapping from the inputs to the corresponding desired output. Once such a mapping has been found, it can be used to provide an assessment of new items for which the desired output is unknown. Obviously the quality of the assessment for unfamiliar items depends to a large degree on a suitable choice of example data: if we choose items which are not representative for the overall set of items, we cannot expect the network to come up with a good assessment for the unfamiliar items.

This second approach can be used to reproduce the assessment resulting from the metrics calculation, e.g by using the calculated value of a metric as desired result in the example set. On the other hand, it can be based on assessments performed by experts, and then provides a way to check the formal definition of a metric against the intuitive judgement of an expert.

3 Intermediate Representation of Programs

Comparing the quality of programs written in different languages based on their source codes is unpractical at the best; this should be done on the basis of a representation which is as independent as possible of the particular language used. We use an intermediate representation (IR) which captures the essential statical and dynamical aspects of a program or large software system, and represents them in an appropriate way, independent from a particular programming language. Figure 3 gives an overview of the generation process for the intermediate representation and indicates some of the particular dependency relations together with their usage; detailed information can be found elsewhere [14, 13].

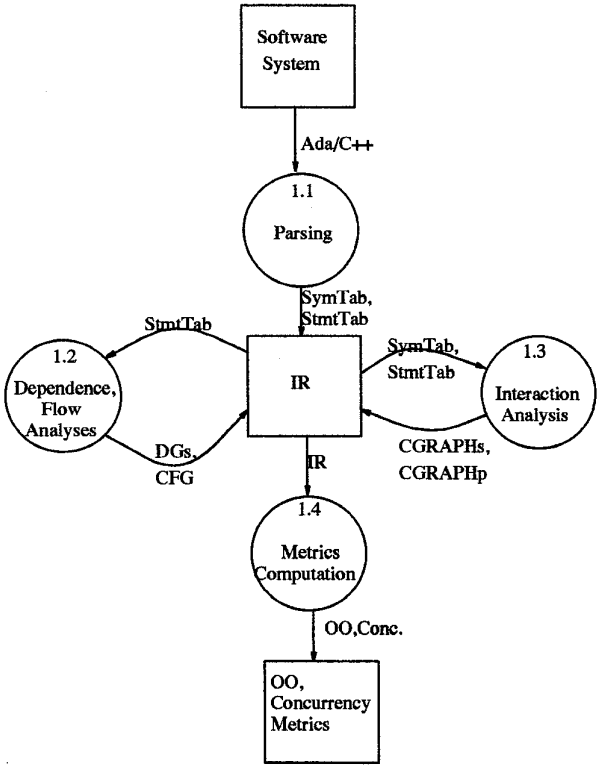

Figure 3: Intermediate Representation

Although this intermediate representation provides an equal basis for the comparison of programs, some quality aspects certainly will reflect particular aspects of the programming language used, as well as program design aspects and programming style. From a reengineering perspective, however, it is largely irrelevant if the poor quality of a program stems from the use of a restricted or unsuitable language, or if it reflects inadequate design or programming practices.

A formal definition of the internal representation described here has been developed in previous papers, e.g. [14, 13]. It is based on the view of software systems as composed of several layers, or tiers. At the highest level considered here, tier 1, are the modules, which may be instances of ADTs that export types and operations (e.g., Ada packages or Modula-2 modules) and/or instances of classes that encapsulate objects and export operations (as in C++, Eiffel and Smalltalk). The modules at tier 1 are composed of tier 2 elements, operations (subprograms or methods). Each operation is composed of a sequence of statements, the elements found at tier 3. At this level of granularity, several important features are captured in the IR. The symbol table (SymTab) and the statement table (StmtTab) [9] are extracted and used for dependence and flow analyses. Dependence analysis involves processing of the StmtTab to extract graphs that represent statement-level precedence relations due to control dependences, data dependences, and code dependences (see Figure 3).

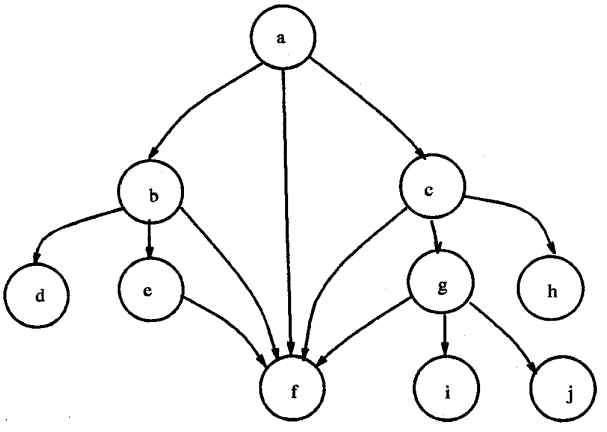

Figure 4: Dependence Graphs

Dependence graphs represent program statements as nodes and use directed edges to denote statement ordering implied by the dependences in a source program.

Different kinds of ordering requirements are represented in different dependence graphs. In the data dependence graph (DDG) a directed edge denotes a data dependence (which means that destination and source nodes need the same variable). The instance dependence graph (IDG) uses undirected edges to denote instance dependences (which occur when two nodes use

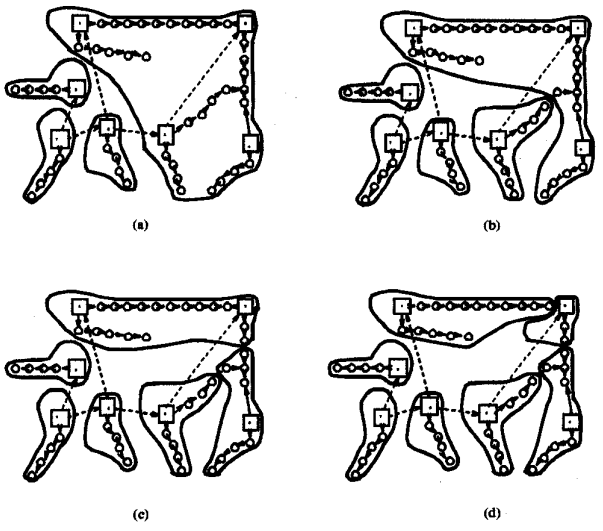

Figure 5: Dependence Graphs and Distribution of Program Components

operations exported by the same instance [11]). The subprogram dependence graph (SDG) uses an undirected edge to denote when two statements use the same subprogram. A directed edge in the control dependence graph (CDG) denotes that execution of the destination statement depends on a decision made by the source statement. In addition to the dependence graphs, the control flow graph (CFG) is extracted at the statement level, indicating the sequential flow of control dictated by the order of the statements in the source code. The analysis of dependencies between system components is also used as the basis for distributing the components of a system among different processing elements.

4 Metrics for Programs

The separation of components describing distinct elements of the real world is an important aspect of object-oriented software design, aiming at systems which are easy to design, understand, implement and maintain. A closer look into properties of such systems reveals the following important aspects:

information hiding (IH) implementation details and design decisions are hidden within modules

cohesion (CH) each module provides a single abstraction

encapsulation (E) related types and operations are grouped within the same module

loosely coupling (CU) there is little or no interdependence among implementations of modules; a change in the implementation of one module should not require changes in the others

Object-orientedness then is computed as a combination the above factors.

Information Hiding This metric measures how well implementation details are hidden from users. data structures should be accessed only via calls to subprograms exported by the module. Thus, a consequence of information hiding are frequent subprogram calls and a deep call graph. Layering is therefore a property observable in the call graph of an application. Average layering is the sum of lengths/depths of all paths from the root to a leaf in the graph, divided by the total number of paths.

Cohesion Cohesion measures the amount of "functional-relatedness" of concepts exported by a module. If an ADT module exports only one type, it is inferred that the module exports a single abstraction and is highly cohesive. If a module exports more than one type, then it is likely to export several concepts; hence, the parts of the module are not highly cohesive. As a consequence, understandability and usability of the module may decrease.

Encapsulation Encapsulation gives a measure for the containment of implementation details within the internal/hidden portions of the modules. It is expressed as a ratio of the number of types and subprograms whose implementation details are encapsulated within modules, to the number of types and subprograms whose implementation details are visible outside of modules.

Coupling Coupling is a numerical measure of the amount of interconnection between software components. This metric should reflect the influence of modifications to one software entity on the correctness of other entities. Thus, the coupling metric is a measure of the amount of interconnection between an application's globally defined data (including module exported data) and its subprograms (including module primitive operations). A high metric value indicates a high amount of interconnection.

Object-orientedness (OO) then can be expressed as a combination of the above metrics. It should be proportional to information hiding, encapsulation and cohesion and inversely proportional to coupling. The formula is:

$$OO = IH + CH + E - CU + 1 \quad (1)$$

The addition of 1 is a correction required by the definition of CU; the range for object-orientedness is between 0 and 4 since each of the terms of the formula has values between 0 and 1.

The maintainability metric is a function of the object-orientedness metric (OO) and the Halstead length metric (HL) (defined in [3]:

$$M = \frac{\mathrm{OO}}{\mathrm{HL}} \quad (2)$$

It decreases ascomponent coupling increases, and as component cohesion, information hiding and encapsulation decrease. Additionally, maintainability of software is inversely proportional to its size.

Exact definitions and techniques to compute metrics for information hiding, cohesion, encapsulation, coupling, object-orientedness and maintainability are described in previous publications [13]. The first four of these metrics correspond to properties found in systems designed in an object-oriented way. The other two, object-orientedness and maintainability, are based on the previous ones, and indicate a combination of desirable properties.

When assessing properties of programs like information hiding, cohesion or object-orientedness, humans frequently – in addition to the particular features relevant for the property – take into account the similarity of the program under investigation to other programs assessed before. Such an approach can be especially useful in cases where there is no obvious formal definition of a property, or when such a formal definition is under development and has to be checked against the intuitive meaning of the property.

We are investigating the use of neural networks for a similarity-based assessment of such properties of programs. In one approach, self-organizing topology-preserving maps [5] are used to categorize the programs under investigation. In another variation, we employ networks capable of generalization, such as backpropagation networks, in order to assess new programs based on previously learned examples. These two approaches and the corresponding experiments are described below.

5 Categorization of Programs

The purpose of this set of experiments is to categorize programs with respect to particular properties or metrics such as object-orientedness, information hiding, cohesion, etc. We use self-organizing networks known as Kohonen networks, self-organizing maps, or topology-preserving maps. Such a network assumes a topolocigal structure among its units, and tries to map similar input patterns onto neighboring units. This type of network consists of a set of n *input units* together with a set of m *cluster units*, wich also are used to display the output of the network. The cluster units are usually arranged in a one- or two-dimensional array; frequent neighborhoods in the two-dimensional case are the rectangular grid (eight neighbors) or the hexagonal grid (six neighbors). Each input unit is connected to all cluster units, and there are no connections among the input units or cluster units themselves. When an input vector is presented to the input units, the cluster unit whose weight vector matches the input pattern best is selected as the winner. During the self-organizing process, the weights of the winning unit and those of its neighbors are then updated in order to provide a better match for that particular input pattern.

In our application, we use a separate Kohonen map for each of the properties or metrics to be investigated. The inputs are a set of features which we consider relevant for that property. The network is presented with all the input vectors of the programs we want to analyze. During the self-organization process, the weight of the cluster units are modified in such a way that similar input patterns are represented by cluster units in the same neighborhood; programs with dissimilar input patterns correspond to distant cluster units.

Experiments In our current series of experiments, we are encountering problems with the clustering of vectors which should be rather similar. Although we cannot say so for sure, we assume that the problem lies with the tool used, the Stuttgart Neural Network Simulator (SNNS) [15]; problems with its usage for Kohonen maps have also been reported on the SNNS mailing list, and seem to be caused by a faulty initialization routine. We are waiting to perform these experiments with a new version of the tool, and are also investigating the use of other tool sets.

6 Learning and Generalization of Metrics

The goal of these experiments is to learn the relationship between the values of a set of features relevant for a particular metric, and the assessment of the program according to either a formula computing the metric, or a human expert. Backpropagation networks are frequently employed for such tasks; they usually consist of an input layer, one or more hidden layers, and an output layer. Connections are present from each unit of the input layer to all units of the hidden layer, and from each unit of the hidden to all units of the output layer; there are no connections within the layers. The network is trained based on a set of example inputs together with the desired output values; in our case we are mainly interested in the generalization capability of such a network: when presented with a new pattern, we would like to get a reasonable output based on the most similar cases learned by the net. A backpropagation net is trained by presenting an input pattern, computing the activations for the units in the network and comparing the activation values for the output units to the desired output values for that pattern. The deviation of the output units' activation values from the target values (also known as error) is then used to modify the weights of the connections in the network.

Experiments In a first step, we use a network with one hidden layer in combination with the backpropagation learning algorithm to generalize the object-orientedness metric from a number of examples. The network has eight input units, five nodes in the hidden layer, and 9 output units. The inputs are used to indicate the factors relevant for the object-orientedness of an Ada program:[1]

- number of modules
- number of types defined outside of modules
- number of subprograms defined outside of modules
- average layering of the call graph of the program
- total number of types exported by the modules in a program
- total number of subprograms exported by the modules in a program
- coupling
- sum of the reciprocal of the number of types exported per module

The number of output units is determined by the granularity of our "object-orientedness scale": the range is from 0 to 4, in steps of 0.5. We used a set of 100 examples, 70 for training purposes and 30 for testing. Here we encountered two problems: First, it was not so easy to find 100 Ada programs suitable for our purposes. Second, even if we have enough example programs, we need an expert to evaluate the programs with respect to their object-orientedness in order to provide the backpropagation network with a target value. Considering the restrictions in our resources, it turned out to be impractical to use real programs, and we decided to generate our test data automatically. This must be done carefully, however, since the input parameters are not mutually independent; a simple generation of random values is not adequate. We developed an algorithm for the generation of these values which on one hand takes into account the dependencies between the input parameters, and on the other hand allows the selection of reasonable values for the independent variable [1]. Learning of the 70 test vectors was typically achieved within a few hundred epochs by vanilla backpropagation algorithm and random initial weights. When applied to our 30 test vectors, we achieved 100% generalization.

[1] Note that this set of parameters is not very sophisticated, and is intended as a starting point to validate the methodology used. One of its main purposes was to expose weaknesses of the approach, which it did.

7 Summary and Conclusions

The goal of this paper has been to outline some experiments using neural networks for the quality assessment of large software systems, in particular with respect to reengineering purposes. One set of experiments uses self-organizing feature maps, or Kohonen maps, to identify and group together programs with similar properties. On one hand, it can help the validation of existing metrics by checking if programs with similar properties also have similar values in their numerical metrics calculations. On the other hand, it can be used to identify novel properties or qualities which have not been captured by the existing metrics yet. The other experiments, using backpropagation networks which learn to map certain input features with outputs reflecting quality assessments, again help with the validation of metrics; one way to do this is to check the assessments based on calculated metrics

against those performed by the network on the basis of a set of examples which has been provided by experts. Another way is to utilize the generalization capability of these networks: the assessments given by the network for new programs which were not in the example set is compared with the assessment based on the calculated metrics. Should there be any difference in the two assessments, it could mean that the set of examples chosen is not really representative, or that the formula used to calculate the metrics might have to be modified. A third possibility is to analyze the internal representation of the network in order to determine the relevance of the particular features used as input; some features might be irrelevant, whereas others might be interdependent.

Acknowledgements

This work has been partially supported by the U.S. NSWC (under contracts N60921-93-M-1912, N60921-94-M-2555, N60921-94-M-1960, N60921-94-M-G096, N00178-95-R-2007, and N60921-95-M-0311), by the U.S. ONR (under contract N00014-92-J-1367), and by the State of New Jersey (SBR-421290 and SBR-421330).

References

[1] G. Dheenadhayalan, *Learning of Software Metrics with Neural Networks*, Masters Thesis New Jersey Institute of Technology, CIS Department, 1995.

[2] L. V. Fausett, *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*, Prentice Hall, 1994.

[3] M. Halstead, *Elements of Software Science*, North Holland, 1977.

[4] J. A. Hertz and A. S. Krogh and R. G. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley, 1991.

[5] T. Kohonen *Self-Organization and Associative Memory*, Springer 1988.

[6] F. Kurfeß. *Parallelism in Logic — Its Potential for Performance and Program Development.* Artificial Intelligence. Vieweg Verlag, Wiesbaden, 1991.

[7] F. Kurfeß. WINA – Knowledge Processing with Symbolic and Sub-Symbolic Mechanisms. Technical report, University of Ulm, Department of Neural Information Processing, D-89069 Ulm, 1993. German version in: KI 93 Workshop on "Wissensverarbeitung mit neuronalen Netzen (Knowledge Processing with neural networks)".

[8] T. J. McCabe, "A Software Complexity Measure," *IEEE Transactions on Software Engineering*, **2(6)**, Dec. 1976, pages 308-320.

[9] B. Ravindran, "Extracting parallelism at compile-time through dependence analysis and cloning techniques in an object-based paradigm," M.S. Thesis, New Jersey Institute of Technology, May 1994.

[10] M. Sitaraman, L. R. Welch and D. E. Harms, "On Specification of Reusable Software Components," *International Journal of Software Engineering and Knowledge Engineering*, World Scientific, **3(2)**, June 1993, pages 207-229.

[11] L. R. Welch, "Cloning ADT Modules to Increase Parallelism: Rationale and Techniques," *Fifth IEEE Symposium on Parallel and Distributed Computing*, pages 430-437, December 1993.

[12] L. R. Welch, A. L. Samuel, M. Masters, R. Harrison, M. Wilson and J. Caruso, "Reengineering Complex Computer Systems for Enhanced Concurrency and Layering," *Journal of Systems and Software*, **30(2)**, pages 45-70, July 1995.

[13] L. R. Welch, M. Lankala, W. Farr and D. Hammer, "Metrics for Quality and Concurrency in Object-Based Systems."

[14] L. R. Welch, G. Yu, B. Ravindran, F. Kurfess and J. Henriques and M. Wilson, "Reverse Engineering of Complex Navy Systems," *International Journal of Software Engineering and Knowledge Engineering*, (to appear).

[15] A. Zell, *Simulation Neuronaler Netze*, Addison-Wesley, 1994.