DIVISION OF COMPUTER SCIENCE

Dynamic Competitive Learning Applied to the Clone Detection Problem

P. Barson
N. Davey
R. Frank
D. S. W. Tansley

Technical Report No 220

February 1995

[illegible]

[illegible]

Abstract

[illegible]

1 Background

1.1 [illegible]

Many large software systems have evolved over a long period of time and have made use of relatively [illegible] software engineering techniques such as "cloning". Here a unit of existing code is adapted for a new purpose, usually by copying and minor modification. Whilst this represents re-use of resources, it can also result in software which is difficult to enhance and maintain. One of the major problems with cloned software in a large system is simply that of locating clones of a unit of code and it is this issue which is addressed in the

Dynamic Competitive Learning Applied to the Clone Detection Problem

P. Barson†, N. Davey*, R. Frank*, D.S.W. Tansley†

†BNR Europe Limited
London Road, Harlow, Essex, UK, CM17 9NA
paulb@bnr.co.uk, dswt@bnr.co.uk
** University of Hertfordshire*
College Lane, Hatfield, Herts, UK, AL10 9AB
comqrnd@herts.ac.uk, comqrjf@herts.ac.uk

Abstract

A novel neural network architecture, 'Dynamic Competitive Learning', has been applied to the software clone detection problem in telecommunications systems. A software clone is a copy (subsequently modified), of a piece of existing code; it is common practice in large systems to obtain similar code via reuse of fully tested code. This paper discusses a hierarchical network with dramatically reduced training time compared to a Self Organising Map, but with at least as good a classification performance.

1 Background

1.1 Clones

Many large software systems have evolved over a long period of time and have made use of relatively unsophisticated software engineering techniques, such as 'cloning'. Here a unit of existing code is adapted for a new purpose, usually by copying and minor modification. Whilst this represents re-use of resources it can also result in software which is difficult to enhance and maintain. One of the major problems with cloned software in a large system is simply that of locating clones of a unit of code and it is this issue which is addressed in the

body of work reported here and in our earlier paper [2]. This task must be accomplished from the source code alone as historical documentation may not be sufficiently detailed in all cases to enable an historical audit to be accomplished.

1.2 Overview

A solution to this problem is, firstly, to represent software units by some numerical feature vector and then to use an unsupervised neural net to identify clusters in these feature vectors. The neural net should also organise the clusters so that it is possible to locate less likely clones than those found in the immediate cluster of the unit being examined.

In view of this latter requirement a natural network architecture to use is Kohonen's self organising map (SOM) [4], in which the clusters found are organised in a two-dimensional topographic map. This was the approach that was taken in our first system, as reported in [2]. Whilst the SOM approach performed reasonably well, in a search for improved performance, a novel neural network architecture, Dynamic Competitive Learning [5] has been investigated.

2 Representation

The requirement here is that a block of source code, written in a standard block structured high level language, should be represented by a numeric vector in such a way that similar blocks of code have similar representational vectors. It is important to note here that the notion of 'similar' is different in either case; for the source code similarity relates to the probability of the code being cloned, and for the corresponding vectors that they are close to in Euclidean space. More specifically we require that if software unit A is represented by vector $\mathbf{v}_A$ and B by $\mathbf{v}_B$ then if B is a clone of A $\|\mathbf{v}_A - \mathbf{v}_B\|_2$ should be relatively small.

In fact the set of source code units can be thought of as a set of indexed taxonomic trees, where the root of each tree corresponds to the first occurrence when a unit was cloned and branching points further down the tree correspond to further instances of cloning — the whole structure is analogous to a set of phylogenetic trees. The index of each tree denotes the degree of divergence of the clone from its parent. Due to the equivalence of indexed hierarchical trees and ultrametric [6] spaces it can be seen that the task of detecting clones becomes one of inducing an appropriate ultrametric in the space of representational vectors. Clustering the source code vectors with a SOM provides a similarity metric, whereas our newer model produces the desired ultrametric - the vectors are organised into a tree structure.

2.1 Representation Vector Definition

To represent all the information in a source code unit would not be feasible due to the resulting size of the vector and the need to represent a variable length

structure in a fixed length format. Some degree of abstraction is therefore required. Our first simplification was that the user chosen tokens (e.g. identifiers or operators), should be largely ignored; this gives a big reduction to the size of the problem. We were therefore left with the problem of capturing the information in the parse tree of the software unit. We do this in three ways: firstly the frequency of keywords in the unit are accumulated, secondly the indentation pattern is represented and lastly the length of each line is recorded.

The method is applicable to any source code language (it has even been suggested that compiled code may be usefully examined).

As all the code is printed in a standard format (or can be easily filtered to a standard format), which is generated by the programming environment, the indentation pattern is isomorphic to the structure of the parse tree. The problem with representing it is that the number of lines in a unit of software, and therefore the number of indentations, is variable. To map this to a fixed length vector we first took the raw indentation values and viewed them as ordinates on a graph; we then sampled one hundred points from this graph, using linear extrapolation where necessary. This coding method is relatively stable against minor modifications to the source code, such as the addition or removal of a line.

The line length gives some indication of the user defined tokens; it was coded in a similar way to the indentation pattern, again being mapped to one hundred points across the software unit.

Finally each field in the vector was normalised, so that each field had roughly the same importance. Each keyword frequency was divided by the maximum frequency for that keyword averaged over a large set of source code units. Each indentation and line-length value was divided by a mean value calculated as before.

3 Results Using the SOM architecture

The results produced in [2] used 1775 input vectors taken from 5 million lines of source code. A SOM was trained using these vectors. At the end of training 100 test procedures were randomly chosen and the clone database was used to access the nearest procedure to it subject to a cloniness threshold. The two procedures were viewed side by side by a software engineer familiar with PROTEL[1] source code who then decided whether the neighbouring procedure was or was not a clone of the test procedure. A total numerical value of success was then calculated by adding the cloniness value if the procedure was a clone and subtracting it if it were not. The final result was then scaled to lie in the range 0 — 100. The best results being 68.7 under this measure.

[1] BNR's proprietary programming language.

4 Dynamic Competitive Learning

4.1 Background

The DCL model [5] differs in two respects from the standard competitive learning paradigm. Firstly the number of nodes in the network is assigned dynamically without any preimposed upper or lower bound and secondly the nodes in the net are organised in a tree structure. The dynamic creation of nodes has been proposed several times in the context of competitive learning and is best known from the ART type networks [3]; the arrangement of nodes in a tree like structure has also been suggested more than once, see [1] for a detailed discussion of this type of method. In this work we make use of the DCL model with a number of refinements identified in the next section.

4.2 Network Specification

The DCL network

The nodes in the DCL network are arranged in a tree structure with a dummy node as root. When an input is presented to the network it is passed to the first layer of the tree, where the standard competition takes place between nodes at this level. If the winner is not sufficiently close to the input, that is the distance is greater than a **quality** value, a new node is created at this level to classify the input. This process is analogous to the use of a vigilance parameter in ART. The quality value determines the radius of the hypersphere, that is the classification volume of a node; this value must therefore decrease as the tree grows deeper, so that lower levels provide a finer classification than higher levels.

If a new node is not required then the winner moves towards the input, mediated by a learning rate, as in standard competitive learning. The learning process is then recursively applied to the children of the winner, until a leaf node of the tree is reached.

At any stage of the learning process a winning node without children may procreate. This occurs if the relative frequency of wins of the node against its parent exceeds a **threshold** value. In order to prevent unbounded growth in the tree the threshold value is increased for lower levels in the tree.

More formally the algorithm is:

Initialise learning rate, r, threshold, t, quality q and Tree to be the root node
Repeat until convergence criterion met
  Set **x** to the next input vector
    Select the child of Tree that is closest to **x**, **w** say
    If | **w** - **x** | < q *then*
        move w towards x, according to **w**' = **w** + r(**x** - **w**)

Increment the win count of **w**
If **w** has children *then* set Tree to **w**, increment t, decrement q
and recurse
else if the win count of **w** / win count of Tree > t
generate for **w** a child with weight vector = **x**
Else
Create a new child of Tree that has weight vector identical to **x**
End Repeat

The shape of the resulting tree is determined by the input vectors and the choice of learning parameters. The following factors must be specified, actual values used are in brackets:

- The quality value of the top level nodes
- The rate at which the quality factor decreases for successive layers as a proportion of the previous value
- The threshold value of the top level nodes
- The rate at which the threshold value increase for successive layers as a proportion of the previous value
- The schedule for the learning rate — which should decrease after each epoch to promote stability as a proportion of the previous value

The Quality Factor was arrived at with the aim of producing a reasonably large initial cluster. With this setting the initial cluster contains 20 nodes with 4 developing a sub classification. So the initial space is divided up into 16 outlier clusters and 4 larger clusters which are sub-divided.

The Quality Reduction rate was chosen so that the network developed the appropriate amount of decomposition and clustering. The very low initial value of the threshold allows the tree to rapidly deepen, but with an increment rate of 1.15, the threshold quickly becomes large: at ten ply the threshold is 0.12, so that a unit must account for 12% of its parents wins before it can gain a successor.

4.3 Training

With these parameters the network converged rapidly; with 10,257 296–ary vectors only two or three epochs were required for reasonable convergence. The training time was less than forty minutes on an HP 712/80.

4.4 The Resulting Network

The network produced consisted of between 7000 and 10,000 units, which even allowing for the redundancy in a tree representation, is obviously excessive. In fact the majority of these nodes were completely redundant; they fell into one of the following two classes:

- Units that did not win during the test phase. These units had been generated during training, but due to migration of nodes in the tree were now redundant
- Runaway branches. Here a branch of the tree would consist of a sequence of only children, so that passage down the branch did not provide any additional classification

When these redundant node were removed the tree consisted of around 1500 active nodes. The shape of the final tree can be characterised as follows:

No of nodes	1500
Maximum Ply	19
Average Branching Factor	2.84
Widest point	298 nodes, at depth 5

The leaf nodes of the tree classified, on average, about 20 input vectors.

5 Results and Evaluation

5.1 Comparison of Classification of SOM and Dynamic network

The dynamic net was tested alongside the equivalent SOM network on the 10,257 vectors.

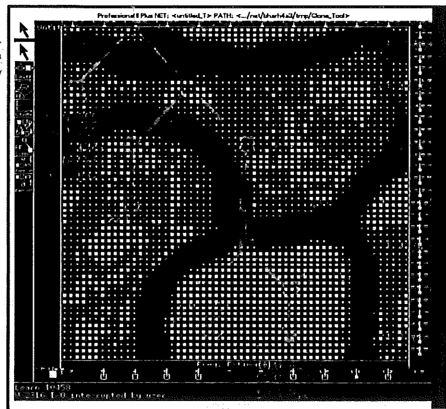

Figure 1. Frequency activation of the nodes contained in the network.

The SOM network used 3025 nodes (55 x 55) with wrap around, an initial neighbourhood of 24 reducing to 1. The SOM produced 4 areas in the output space (see Fig 1).

Similar results were obtained from both networks. Both networks allow the user to navigate their way through the clusters formed; in the SOM by looking at increasing neighbourhoods and in the tree by looking at relations at progressively greater distance, that is increasing the ultrametric neighbourhood.

5.2 Evaluation of the Dynamic Network as a detector of clones

In order to test how well the network found clones, procedures from the test set were modified in ways that a software engineer might do if s/he was cloning the code. The changes made were:

- Adding comment lines
- Changing the indentation by increasing the left hand margin
- Adding a operator, such as 'bind'
- Adding an additional single line statements
- Adding an additional block structures such as 'IF...THEN...ENDIF'

With all of these changes, except the addition of a blocked statement, the classification in the net remained unchanged. The addition of the blocked statement caused a major movement of the classification.

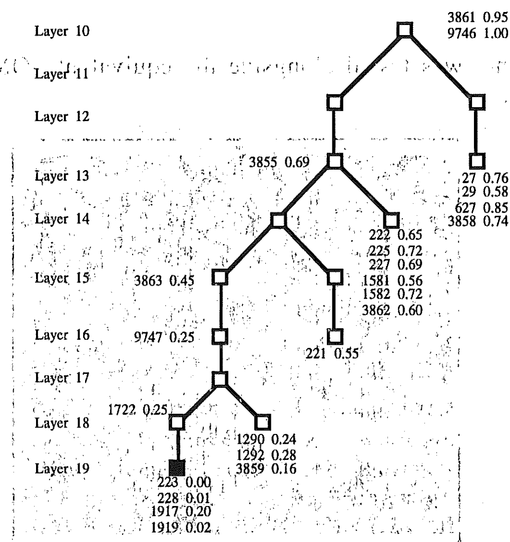

Figure 2 Classification performed by the DCL net.

Figure 2 shows the input (left) and the Euclidean distance (right) from a given test input. For the test input 223 procedure 228 is the closest.

5.3 Conclusions

Generally the DCL model performs as accurately as the SOM on this classification problem and has some significant advantages:

- Training/Recall time: the DCL takes roughly half the time to train and recall is ten times faster
- Size: The trained DCL is about 4Mb and the SOM 20Mb
- Quality of classification: where a tree like structure is anticipated from the classification, i.e. an ultrametrics is expected, it is an obvious benefit for this type of structure to be produced, rather the flat metrics topology of a SOM.
- Incremental training is more resilient; as additional areas of the tree are generated, rather than occupying previously assigned areas as in the SOM.

Using a DCL can be difficult however. The network is very sensitive to the selection of parameters, and obtaining the optimal setting can be problematic. It is important to remove redundant nodes as these constitute a significant part of a resulting net.

Acknowledgements

We gratefully acknowledge the financial assistance of the Department of Trade and Industry of the UK Government, partly funding the placement of Paul Barson on a Teaching Company Scheme at BNR Europe Limited, through the Teaching Company Directorate organisation, under grant no. TCS–1326.

References

[1] K. Butchart, "A Comparative Study of two Self Organising and Structurally Adaptive Neural Tree Networks," *University of Hertfordshire Technical Reportt* ,1994.

[2] S. Carter, R.J. Frank, & D. S. W. Tansley, "Clone Detection in Telecommunication Software Systems: A Neural Network Approach," *Proceedings International Workshop on Applications of Neural Networks to Telecommunications* , pp273-280,1993.

[3] G. Carpenter, & S. Grossberg, "The Art of Adaptive Pattern Recognition by a Self-Organising Neural Network", *IEEE Computer* , Vol 21(3), pp77-88. 1988.

[4] T. Kohonen, "The Self-Organising Map", P*roceedings of the IEEE*, Vol. 78, No 9, 1990.

[5] J. Racz, & T. Klotz, "The Dynamic Competitive Learning Method", *Computers in Industry*, 17, pp155-158, 1991

[6] R. Rannal, G. Toulouse, & M. A. Virasoro, "Ultrametricity for Physicists", *rev. mod. phy.*, 58, pp765-788, 1986.