ANCHOR - A Connectionist Architecture for Hierarchical Nesting of Multiple Heterogeneous Neural Nets

Rajeev Kumar and Peter I. Rockett

Autonomous Systems Group
Department of Electronic & Electrical Engineering
University of Sheffield
Mappin Street, Sheffield S1 3JD, England
{r.kumar, p.rockett}@sheffield.ac.uk

Abstract

We present a novel connectionist architecture for handling arbitrarily complex neural computations. ***A***rtificial ***N***eural Network ***C***ompiler for **H**ierarchical ***OR***ganisation (*ANCHOR*) takes a generalised view of a neuron processing element which we have termed as "Superneuron"; and which accepts n-inputs and outputs m-features where m=1 for an element in a conventional network. A superneuron is a single or higher-order multi-layer perceptron which can be trained individually with support of multiple-learning algorithms from the system. The indistinguishability between a superneuron and a neuron is employed in hierarchical nesting of superneurons up to (theoretically) infinite depth within other superneurons as well as linear or tree-structured cascading. *ANCHOR* facilitates partitioning of input space into a number of small simpler sub-mappings and has been tested on learning simple boolean functions with hierarchical calls and cascading of simpler superneurons. The issues related to partitioning and self-partitioning for a statistical hierarchical classifier are discussed.

Keywords: Hierarchical neural network, Connectionist architecture, Superneuron, Data-partitioning.

Introduction

Ongoing technological advances in connectionist intelligence and their increasingly stronger links with statistics are contributing towards the distributed learning of multiple nets accompanied by their integration into hierarchical/modular networks. Such an integration is attractive for problem-solving in complex domains of neural computations driven by a desire for - (i) faster solutions and convergence, (ii) fewer misclassification errors or more accurate predictions, and (iii) better mappings on coarse-grain multiprocessors. There is always a need of test-beds to explore different network architectures and paradigms. Though there exist many simulation environments for experimenting with monolithic networks, the research activity in modular systems is hindered due to the lack of general purpose modular system environment(s).

One of the very first modular systems, *Mod 2* (Mumford, Andes, & Kern 1992) is a layered architecture configured around special purpose neural hardware. Other simulation environments exist, *e.g.*, *Sesame* (Linden & Tietz 1992), *MUME* (Jabri, Tinker, & Leerink 1992) and *MONNET* (Lange & Manner 1993) for combining multiple neural network paradigms. They are different in their design, but in principle they all have the same functionality, *i.e.*, all combine similar or different net architectures in layered cascading. In this paper we present *ANCHOR* - a connectionist architecture which we have developed for integrating multiple heterogeneous neural networks combining hierarchical nesting in addition to layered cascading. Hierarchical nesting of networks is the main strength of *ANCHOR* and supports, for example, recursive partitioning of input feature space, pre-processing and embedding of *a priori* knowledge.

Modular System Strategy

ANCHOR facilitates solutions to both network modularisation and data partitioning. On modularity, *ANCHOR* is designed to integrate arbitrarily many heterogeneous neural nets in both hierarchical nesting and cascading along with non-neural processing modules. Additionally, a neuron in a net can also possess higher-order connections. The input signal within the *ANCHOR* framework is propagated just like a conventional feedforward neural network. The errors are backpropagated essentially in the standard backpropagation learning algorithm paradigm of (Rumelhart, Hinton, & Williams 1986) but with a distinction that they are backpropagated through the net hierarchy. Since each backpropagation learning algorithm has its own advantages and drawbacks and is suitable for particular training data-sets, *ANCHOR* supports multiple learning algorithms with an extensibility where a user can also code his/her own algorithms. Multiple instantiations of a network is a common requirement; the notion of modularity demands that even similar networks learn different training patterns, and thus learn different

mappings. This property has implications in learning and to the forms of generalisation where multiple instantiations are left with different net-topology and connection-strengths. They can be decompiled independently within the *ANCHOR* framework for future recall. *ANCHOR* has an in-built **N**et **D**efinition **L**anguage (NDL) syntax to define the net and its hierarchy, cascading and connectivity; the system may decompile a trained network into NDL.

Whereas modularity addresses the drawback of scalability of neural networks and its use to solve complex problems the *credit assignment problem* (i.e., the problem of getting the right training information to the right synapses in a network so as to improve overall system performance) becomes more complex and adds another problem-dimension of getting the right information to the right module. This implies that the modularity can serve meaningfully only if both the inter-module as well as intra-module assignments are beneficial to credit assignment. Modularity is basically a divide-and-conquer approach of dividing the problem into a series of sub-problems, assigning a set of function approximators to each sub-problem so that the modules learn to specialise in different tasks and then combining their individual solutions. (Jacobs, Jordan, & Barto 1991) inferred that function decomposition is an underconstrained problem and different modular architectures may decompose a function in different ways which is certainly not a happy situation from generalisation point of view of a neural network with too many degrees of freedom. They also concluded that the modular architecture can be restricted to a well-suited solution if domain knowledge is incorporated for a desirable decomposition. They (Jordan & Jacobs 1992) introduced the design of hierarchical mixtures of experts where expert networks compete to learn the training patterns, and the gating network mediates the competition. Their architecture performs task decomposition in the sense that it learns to partition a task into functionally independent subtasks and allocates a distinct network to learn each task. In subsequent years, different researchers have used different ways of incorporating *a priori* knowledge into modular architecture for both functionality of task-decomposition and modular learning integrated together.

Adopting a different line of research, we, in *ANCHOR*, are trying to withdraw the task of decomposition from the regime of modular learning. We are attempting input space partitioning in a generic way, where a user can code application-specific knowledge in multiple-objective functions and the input space is recursively partitioned into feature subspaces in a near-optimal manner. The process of data-partitioning acts as a pre-processor to the modular network and the success rate of modularity depends how effectively *a priori* knowledge is encoded into the decomposition strategy, and their corresponding mappings on the networks of appropriate architectures. This strategy of pre-processing the data before its submission to the neural domain considerably reduces the learning-complexity.

In conceptualisation, *ANCHOR* has strong links to the recursive feature space partitioning algorithms of (Henrichon & Fu 1969) and (Friedman 1977) for non-parametric classification using hyperplanes parallel to feature axes. The rationale for such hierarchical partitioning, and their interactions and differences between artificial intelligence and pattern recognition are summarised by (Kanal 1979). In another simpler version of the algorithm, (Sethi & Sarvarayudu 1982) have shown that the algorithm is suitable to a real multi-feature multi-class problems involving handprinted numeral recognition. Similarly, *ANCHOR* aims to partition feature space into subspaces (although not necessarily with hyperplanes) and their corresponding mappings on hierarchical neural networks for efficient problem solving.

System Architecture

ANCHOR encapsulates four sub-systems illustrated in figure 1 - superneuron (network) instantiation, user-NDL interface, (optional) data-partitioning & optimiser, and (optional) non-neural computation.

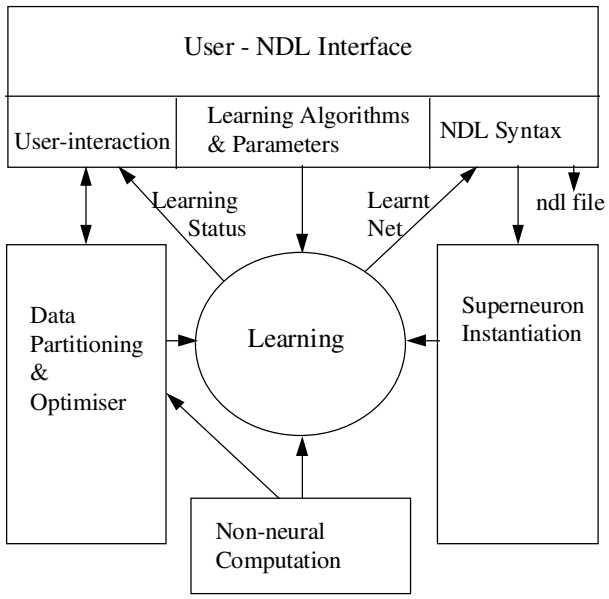

Figure 1: *ANCHOR* System Architecture

Superneuron Instantiation

The whole design of *ANCHOR* evolves around the "***Superneuron***" concept, which we have defined as the generalisation of a neuron processing element. A superneuron accepts n-inputs and outputs m-features where as m=1 for an element in a conventional network; a superneuron is a conventional (single-order) or a higher-

order multi-layer perceptron (MLP) - figure 2. This generalisation of a neuron, *i.e.* indistinguishability between a neuron and superneuron, empowers *ANCHOR* to integrate multiple heterogeneous neural nets:

- a neural network (MLP) can be trained individually or co-operatively on some subspace;
- a trained net (MLP) can be treated again as a neural element with m≥1, *i.e.* a superneuron;
- a MLP can connect many neurons (superneurons) in its multi-layers which can be treated as layered cascading of MLPs, but this is again a superneuron; and
- a superneuron can be connected in a layer of a MLP similar to that of a neuron. This situation results in a nesting of network(s) within another network; again this is a superneuron, and so on..

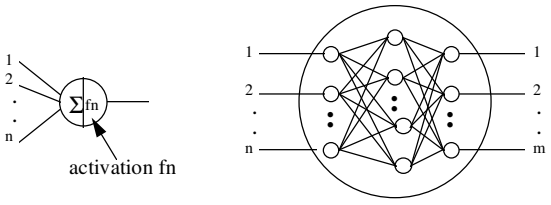

Figure 2: Generalisation of neuron *i.e.* a superneuron

The instantiation of a superneuron at the outer-most level is treated as 'global' to distinguish it from the 'nested' instantiations. An instance of a superneuron can be a 'learn' or 'recall' type, depending on whether it is to be trained or recalled, respectively. A symbol table is maintained for each superneuron, and it stores the information related to type of node, e.g., input node, simple neuron, superneuron or output node. The entries in symbol table are used in mapping the links among superneurons for feeding forward the signal and backpropagating the errors through them. Within a superneuron, single(higher) order connection fields for each simple neuron are established and the pointer(s) to the input signal is(are) referenced. Each global superneuron is associated with a stack onto which all the nested superneurons are pushed during 'call' routines; and are popped while establishing the appropriate links among them.

User-NDL Interface

The complex situation of superneuron instantiation is made simple by the Net Definition Language (NDL). The syntax is simple and short, and the Extended Backus-Naur Form (BNF) (Aho, Sethi, & Ullman 1986) of the grammar is given in the appendix. A user defines only the net topology and connectivity in a simple text description. The NDL file is lexically examined for the correct syntax and is then compiled to either executable code or a *C++* program. The example in the section "Implementation" briefly explains the definition of a nested and cascaded network in NDL syntax.

The compilation of a NDL file results in instantiation of the "global" superneuron. A user interface is provided for the selection of the learning algorithm and parameters, inputting the training, validation and/or test pattern sets, feeding training-stopping criterion, monitoring the status of learning, and prompting for pruning and de-compile/overwrite options.

Data-Partitioning and Optimiser

We briefly discuss the problem of determining which of the sub-feature spaces are optimal for partitioning the pattern space and the corresponding mappings onto neural networks. Genetic algorithms have been used for this partitioning to improve classification performance and reduce computation (Chang & Lippmann 1991). A class of classifier systems and their possible extension through genetic algorithms as a *rule discovery system*, and determination of an effective subset of rules for improving the overall system performance through *credit assignment system* is reviewed in (Booker, Goldberg, & Holland 1989). We suggest an approach based on multi-objective genetic algorithms (Fonseca & Fleming 1993) which has the benefit that the partitioning strategy can accommodate multiple and possibly competing objectives.

At the moment, we envisage that the multi-objective genetic algorithms search techniques will decompose the input space into the clustering hyperspheres, which can be potentially partitioned again into smaller hyperspheres resulting into a near-optimal partitioning. The statistical pattern recognition literature (Fukunaga 1990) contains a great deal of work of selection of optimal features from such subspaces - an overview of many different approaches is given in (Siedlecki & Sklansky 1988). We may apply certain transformations/heuristics for the selection of dominant features in each of the hyperspheres reducing considerably the data-dimensions for final processing through the neural networks (Brill, Brown, & Martin 1992).

Alternatively, an applicable non-parametric technique for estimating a class-conditional probability density function is multivariate Parzen windowing although choice of scale factors has a significant influence on its performance. The clustered data can then be 'fused' within the *ANCHOR* framework to approximate a Bayesian classifier (Richard & Lippmann 1991).

These topics are areas of active research.

Incorporation of Non-neural Computation

A superneuron in a network behaves like black-box which receives inputs and produces outputs. The class of superneuron is made generic by adding a field where a superneuron is treated as a 'neural' or a 'non-neural' entity. In the latter case, a reference to a user-defined

(non-neural) routine performs the computation such as linear transformation of the pattern space or mapping to a new basis. This facility has advantages where the pre-/ post-processing of the data is necessary and also permits the possibility of the incorporation of traditional rule-based AI processing.

Implementation

ANCHOR is implemented in *C++* for obvious advantages of object oriented programming. The NDL file is lexically analysed and parsed through *lex/yacc* compiler writing tools. *ANCHOR* functionality has been demonstrated on learning simple boolean function. It demonstrates that the *ANCHOR* concept works perfectly. A case study of learning a "full-adder" logical function is given here. At the outermost level, the superneuron "fulladd" is defined with calls to four other simpler superneurons; one of them, in turn, calls two nested superneurons:

```
// fulladd.ndl              // ndl description of "fulladd"
netdef  fulladd;
// global definition
input = {1..3};             //input node ids 1, 2 & 3
// calls to superneurons "halfadd", "xor", "and" and
//"or" with corresponding input & output node ids.
call halfadd  ({1,2}, {4,5});
// here superneuron "halfadd" is called with input node
// ids as 1&2; while corresponding outputs are mapped
// onto ids 4 & 5. Superneuron connectivity & weight
// strengths are treated as local parameters. Similar
// notations are used for other calls of superneurons.
call xor   ({4,3}, {6});
call and  ({4,3}, {7});
call or    ({7,5}, {8});
output = {6,8};             //output node ids 6 & 8
enddef fulladd;

// halfadd.ndl              // ndl description of "halfadd"
netdef halfadd;             //cascaded
input = {1,2};
call xor   ({1,2}, {3});
call and  ({1,2}, {4});
output = {3,4};
enddef halfadd;

// xor.ndl                  // ndl description of "xor"
netdef xor;                 //multiple instantiation
input = {1,2};
layer = {3,4};              // input layer, neuron ids 3 & 4
// sigmoid is the default activation function
// hidden layer(s) to be listed, if any
layer = {5};                // output layer
output = {5};               // output node
// connectivity for neuron id 3 is given now
b[3] ;                      //bias or zeroth order-connection
w[3,1]; w[3,2];             //first order connections
w[3,1,2];                   //second order connection
// connectivity for neuron id 4
b[4];
w[4,1]; w[4,2]; w[4,1,2];
w[5,3]; w[5,4];             //links for neuron id 5
enddef xor;

//similarly the other superneurons "and" and "or" are
//defined individually.
```

Once a network is trained, it is decompiled for recall, and the NDL file can be overwritten with the connections and their strength. If a network is trained & pruned for generalisation, the layer definitions may also be updated along with the connections; sparsity is permitted. The multiple instantiations of a superneuron are separately decompiled to allow different structures.

The decompiled version of superneuron "fulladd" is given as follows:

```
// fulladd.ndl
// ANCHOR de-compiler output
netdef  fulladd;
// global definition
input = {1..3};
call halfadd  ({1,2}, {4,5});
call xor1   ({4,3}, {6});
// multiple instantiation of superneuron "xor"
// incremental suffixing in superneuron_id "xor"
call and  ({4,3}, {7});
call or    ({7,5}, {8});
output = {6,8};
enddef fulladd;

// halfadd.ndl
// ANCHOR de-compiler output
netdef halfadd;
input = {1,2};
call xor   ({1,2}, {3});
call and  ({1,2}, {4});
output = {3,4};
enddef halfadd;

// xor.ndl
// ANCHOR de-compiler output
netdef xor;                 // first instantiation
input = {1,2};
layer = {3,4};
layer = {5};
output = {5};
// connection links and strength
b[3] = 0.0;
w[3,1] = 2.33906;
w[3,2] = 2.34235;
w[3,1,2] = -3.64402;
b[4] = 0.0;
w[4,1] = -2.68276;
w[4,2] = -2.68130;
w[4,1,2] = 5.71754;
```

```
w[5,3] = 3.76964;
w[5,4] = -11.3740;
enddef xor;

// multiple instantiations are de-compiled separately.

// xor1.ndl
// ANCHOR de-compiler output
netdef xor1;            // second instantiation
.............
.........
enddef xor1;

// similarly the other superneurons "and", "or" and their
// multiple instantiation(s), if any, are de-compiled.
```

Learning a logical function does not necessitate a separate de-compilation of the multiple instantiations, because all the entries of a truth-table are learnt during the process of network training, no novel instances are encountered during recall phase, and there is no existence of overfitting of a network on overtraining. However, this is an important feature and has major applications to a wider range of problem solutions. When networks of initially similar architecture are trained on different data-sets and monitored for out-of-sample performance, they all have potentially different net-topology and connection strengths on pruning for generalisation. Multiple instantiation within a modular network is a common occurrence, and their independent de-compilation after training is the solution under such circumstances.

Conclusions

The *ANCHOR* framework facilitates the definition, partitioning and training of complex neural systems and has been developed to cope with trends in neural computations. Networks can be constructed in hierarchical, cascaded or tree-type fashion or any composition of these, and non-neural processing blocks can be seamlessly incorporated. Key to *ANCHOR* is the concept of a 'superneuron', a generalisation of the conventional neuron. Hierarchical decomposition of simple boolean functions has been demonstrated as proof-of-concept. Future work on *ANCHOR* will focus on the partitioning of high-dimensional pattern spaces as a pre-processing of the input space through encoding *a priori* knowledge before its presentation to neural networks. The recursively partitioned feature-spaces can then be directly mapped onto the hierarchical organisation of superneurons; and thus significant performance improvement is expected.

Acknowledgements

We wish to thank the anonymous reviewers for their valuable comments that prompted some additions. One of us (RK) gratefully acknowledges the support of a Commonwealth Scholarship Commission research studentship.

Appendix

ANCHOR Network Definition Language (NDL) Syntax in extended BNF Notations is given here:

```
NetArchitecture ::= <NetHead>
                    <InputDef>
                    <LayerDef>
                    <OutputDef>
                    {<ConnectDef>}?
                    <NetTerm>
// Each superneuron "global" or "nested" is defined by a
// net-architecture.

NetHead ::= netdef <NetId> ';'
// Net header with an unique net-identifier.

InputDef ::= {input '=' '{' <NodeList> '}' ';'}+
// It contains the Input node ids.

LayerDef ::= <NeuronLayer>
           | <SuperNeuron>
           | <LayerDef> <NeuronLayer>
           | <LayerDef> <SuperNeuron>
// LayerDef defines the layer(s) of the net. A layer can
// consist of simple neurons or superneuron. A net-
// architecture is a single/multi-layered.

NeuronLayer ::= layer '=' '{' <NodeList> '}' [sl]? ';'
// The term 's' or 'l' defines the type of activation function
// for the layer - 's' stands for sigmoid function and 'l' as
// the weighted linear sum. Sigmoid is the default
// activation function.

SuperNeuron ::= call <NetId> '(' '{' <InNodeList> '}' ','
                '{' <OutNodeList> '}' ')' ';'
// Call to a "nested" superneuron with external links to
// input and output nodes. The NDL file "<NetId>.ndl"
// defines the net-architecture. The input and output nodes
// are appropriately mapped.

OutputDef ::= {output '=' '{' <NodeList> '}' ';'}+
// It gives the Output Node ids.

ConnectDef ::= <WeightDef>
             | <BiasDef>
             | <ConnectDef> <WeightDef>
             | <ConnectDef> <BiasDef>
// Simple neuron-connections within a superneuron.

WeightDef ::= {f}? w <WeightConnect>
// The connection strength can be fixed or it can be
// modified during training. The prefix 'f' indicates
// "fixed" connection.
```

BiasDef ::= {f}? b <BiasConnect>
// Bias to a neuron.

WeightConnect ::= '[' <DestNode> ',' <SrcNodeList> ']'
{ '=' <Float> }? ';'
// Source Node List defines the connection order.
// In case of single order connection, it has one element.
// The strength can be randomised depending on the initial
// state.

BiasConnect ::= '[' <DestNode> ']' { '=' <Float> }? ';'
// Bias-term of a neuron.

NetTerm ::= enddef <NetId> ';'
// Net-definition terminator.

NetId ::= <Identifier>

Identifier ::= [A-Za-z_][A-Za-z_0-9]*

NodeList ::= <Node>
| <NodeList> ',' <Node>

Node ::= <DigitSeq>
| <DigitSeq> '..' <DigitSeq>

InNodeList ::= <NodeList>

OutNodeList ::= <NodeList>

SrcNodeList ::= <NodeList>

DestNode ::= <DigitSeq>

DigitSeq ::= [0-9]+

Float ::= [-+]? <DigitSeq> <Exponent>
| [-+]? <FloatDot> {<Exponent>}?

Exponent ::= [eE][-+]?<DigitSeq>

FloatDot ::= <DigitSeq> '.' <DigitSeq>
| '.' <DigitSeq>

References

Aho, A. V.; Sethi, R.; and Ullman, J.D. 1986. *Compilers - Principles, Techniques and Tools*. Reading, Mass.: Addison-Wesley.

Brill, F. Z.; Brown, D. E.; and Martin, W. N. 1992. Fast Genetic Selection of Features for Neural Network Classifiers. *IEEE Trans. Neural Networks* 3 (2): 324-328.

Booker, L. B.; Goldberg, D. E.; and Holland, J. H. 1989. Classifier Systems and Genetic Algorithms. *Artificial Intelligence* 40: 235-282.

Chang, E. I.; and Lippmann, R. P. 1991. Using Genetic Algorithms to Improve Pattern Classification Performance. In Lippmann, R. P.; Moody, J. E.; and Touretzky, D. S. (Eds.) *Advances in Neural Information Processing System 3*: 797-803. San Mateo, Calif.: Morgan Kaufmann.

Fonseca, C. M.; and Fleming, P. J. 1993. Genetic Algorithms for Multiobjective Optimisation: Formulation, Discussion and Generalisation. In Proceedings of Fifth International Conference on Genetic Algorithms, 416-423. San Mateo, Calif.: Morgan Kaufmann.

Friedman, J. H. 1977. A Recursive Partitioning Decision Role for Nonparametric Classification. *IEEE Trans. Computers* 26 (4): 404-408.

Fukunaga, K. 1990. *Introduction to Statistical Pattern Recognition*. San Diego, Calif.: Academic Press.

Henrichon, E. G.; and Fu, K. S. 1969. A Nonparametric Partitioning Procedure for Pattern Classification. *IEEE Trans. Computers* 18 (7): 614-624.

Jabri, M. A.; Tinker, E. A.; and Leerink, L. 1992. MUME - An Environment for Multi-net and Multi-architecture Neural Simulation. Technical Report, Systems Engineering and Design Automation Laboratory, Sydney Univ.

Jacobs, R. A.; Jordan, M. I.; and Barto, A. G. 1991. Task Decomposition through Competition in a Modular Connectionist Architecture: the What and Where Vision Tasks. *Cognitive Science* 15: 219-250.

Jordan, M. I.; and Jacobs, R. A. 1992. Hierarchies of Adaptive Experts. In Moody, J. E.; Hanson, S. J.; and Lippmann, R. P. (Eds.) *Advances in Neural Information Processing System 4*: 985-992. San Mateo, Calif.: Morgan Kaufmann.

Kanal, L. N. 1979. Problem-Solving Models and Search Strategies for Pattern Recognition. *IEEE Trans. Pattern Analysis Machine Intelligence* 1 (2): 193-201.

Lange, R.; and Manner, R. 1993. MONNET: a Software System for Modular Neural Networks Based on Object Passing. In Proceedings of SPIE: Applications of Artificial Neural Networks IV, 1965: 232-243.

Linden, A.; and Tietz, C. 1992. Combining Multiple Neural Network Paradigms and Applications using Sesame. In Proceedings of the International Joint Conference on Neural Networks II: 528-534. Los Alamitos, Calif. : IEEE Computer Society Press.

Mumford, M. L.; Andes, D. A.; and Kern, L. R. 1992. The Mod 2 Neurocomputer System Design. *IEEE Trans. Neural Networks* 3 (3): 423-433.

Richard, M. D.; and Lippmann, R. P. 1991. Neural Network Classifiers Estimate Bayesian *a posteriori* Probabilities. *Neural Computation* 3 (4): 461-483.

Rumelhart, D. E.; Hinton, G. E.; and Williams, R. J. 1986. Learning Representations by Back-Propagating Errors. *Nature* 323: 533-536.

Sethi, I. K.; and Sarvarayudu, G. P. R. 1982. Hierarchical Classifier Design Using Mutual Information. *IEEE Trans. Pattern Analysis Machine Intelligence* 4 (4): 441-445.

Siedlecki, W.; and Sklansky, J. 1988. On Automatic Feature Selection. *International Journal of Pattern Recognition and Artificial Intelligence* 2 (2): 197-220.