A Way to Incorporate Neural Networks into Expert Systems

F. Franek and I. Bruha
Dept. of Comp. Sci. and Systems
McMaster University
Hamilton, Ont.
L8S 4K1 Canada

Abstract

It is well known that both major directions of AI research Neural Networks and Expert Systems exhibit their strengths and weaknesses in almost complementary way. While neural networks are not good in higher-level reasoning (mainly for lack of proper representation as well as proper training methods), they are very good in imprecise classification and recognition. Expert systems on the other hand are reasonably good in higher-level reasoning, but they are fundamentally weak in handling imprecise and uncertain knowledge and data.

This weakness of expert systems in handling uncertainty has been addressed by many researchers and many methods have been proposed to deal with it. We have also contributed towards the solution of this problem by designing and building a software tool **McESE** to help create expert systems. We had had several objectives on our mind while designing the system, but one of them was to give the knowledge engineer a possibility to deal with uncertainty in different and more flexible ways and not to be "locked" in any single approach.

McESE knowledge base consists of a set of rules in the form

$$\mathbf{T_1 \& T_2 \& T_3 \& ... \& T_n =F=> T}\ .$$

CVPF (*certainty value propagation function*) **F** takes as arguments the certainty values of $\mathbf{T_1}$, ... , $\mathbf{T_n}$ and returns the certainty value to be assigned to **T**. Thus, each rule contains not only a declarative knowledge, but also a procedural knowledge (the cvpf **F**) of how to handle and propagate uncertainty within the rule. **F** can be a program written by the knowledge engineer, or it can be any of the predefined functions for emulating standard methods. Each rule can have its own cvpf, or several rules can share one.

While experimenting with McESE and building sample expert systems, we had realized that it was always easier to state the declarative part of a rule than to give the corresponding cvpf. As a solution we are presenting a way to incorporate neural networks in McESE-built expert systems to emulate the cvpf's. This combination separates and preserves the strong aspects of both methods, and complements them in their weak aspects, i.e. the rules are still used to capture the high-level relationships for reasoning, but the knowledge engineer is freed from the detail bother of the dealing with uncertainty, while the neural networks involved do not have to be trained for high-level tasks, but they are trained to "fuzzyfy" the terms in the rules.

1. Introduction

It is well known that both major directions of AI research Neural Networks and Expert Systems exhibit their strengths and their weaknesses in almost complementary way. While neural networks are not good in higher-level reasoning (mainly for lack of proper representation as well as proper training methods), they are very good in imprecise classification and recognition. Expert Systems on the other hand are reasonably good in higher-level reasoning (we are taking the liberty to call the explicit interpretation of rules usually done by inference engines of expert systems *reasoning*), but they are fundamentally weak in handling imprecise and uncertain knowledge and data.

This weakness in handling uncertainty has been addressed by many researchers and many methods have been proposed and implemented to deal with this problem (see e.g. [HH], [G], [J], [P], [W]). We have also contributed a bit towards the solution of the problem of handling uncertainty. Our approach was more practical than theoretical, mainly because most of the main methods are rigorous on its surface only, in a sense they are more ad-hoc methods than well-founded theories (see [J]). We have designed and built a software tool to help create expert systems, so-called **McESE** (*McMaster Expert System Environment*) which we presented at the Zurich conference on Expert Systems in June 1989 (see [FB1], [FB2], [F]). We had had several objectives on our mind while designing the environment, but one of them was to give the knowledge engineer a possibility to deal with uncertainty in a more flexible fashion and be able to select the most suitable approach for the problem at hand, rather than to be "locked" in any single approach.

McESE knowledge base consists of a set of rules in the form

$$\mathbf{T_1 \ \& \ T_2 \ \& \ T_3 \ \& \ ... \ \& \ T_n \ =F=> \ T}$$

where $\mathbf{T_1}$, ... , $\mathbf{T_n}$ and **T** are so-called McESE terms, which for the sake of brevity of the introduction can be described as predicates. CVPF (*certainty value propagation function*) **F** takes as arguments the certainty values of $\mathbf{T_1}$, ... , $\mathbf{T_n}$, and returns the certainty value to be assigned to **T**. Thus, each rule contains not only a declarative knowledge (the relations among different terms), but also a procedural knowledge (the cvpf **F**) of how to handle and propagate uncertainty within the rule. **F** can be a program written by the knowledge engineer, or it can be any of the predefined functions for emulating standard methods (see [J]). Each rule can have a different cvpf, or several rules (or all rules) can share one.

While experimenting with McESE and building sample expert systems, we had realized that it was always easier to state the declarative part of a rule (i.e. the relationship among terms) than to give the corresponding cvpf. As a solution we proposed to incorporate neural networks in McESE-built expert systems to emulate the cvpf's. This combination separates and preserves the strong aspects of both methods, and complements them in their weak aspects, i.e. the rules are still used to capture the high-level relationships among terms for reasoning, but the knowledge engineer is freed from the detail bother of the dealing with uncertainty, while the neural networks involved do not have to be trained for high-level tasks, but they are trained to "fuzzyfy" the terms in the rules.

The project is now in its initial phase, when a complete McESE-built expert system (with cvpf's and all) is used as the "teacher" to a "student" expert system having the same rules, but without the cvpf's. As the "teacher", an expert system to play a card game Canasta was chosen for many reasons. Firstly we needed a non-trivial expert system with enough "impreciseness" to guarantee a broad use of cvpf's. Secondly, as usual, we needed a domain with dedicated human experts at hand (in this case it is us). And thirdly, we needed a lot of input data for training of neural networks - repeated runs of the "teacher" expert system generate as much input data as we desire.

During the first phase the "student" expert system is collecting numbers (certainty values) produced for each rule by the "teacher" expert system during the games, and when the "teaching" phase (i.e. all games) is over, neural networks of the "student" expert system are trained using the numbers collected. The main goal of this stage is to determine the optimal values of various parameters (like initial weights range, gain term, number of hidden layers, number of nodes in a hidden layer) of neural networks involved, since there is not enough theoretical knowledge to preset them before the actual games, and certain heuristics must be used (see e.g. [B], [BM], [H]).

The next phase follows a different strategy of learning for the "student" expert system. At this point the morphology of the neural networks of the "student" expert system as determined during the first stage is fixed. The "student" expert system collects not the certainty values as produced by the "teacher" expert system, but just relative ranking of terms (conclusions) involved. After the training games are over, the "student" expert system has collected a set of inequalities with variables and numbers - they represent rankings of different terms (conclusions) produced during the games. Linear programming technique is used by the "student" expert system to find a solution for the set of inequalities. Note, that this set of inequalities has a solution, namely the numbers as produced by the "teacher" expert system. Thus, we are assured at this stage of successfully resolving the set of inequalities and the solution is then used to train the neural networks as in the first stage.

The third phase is a modified phase two, where the "student" expert system is not able to "watch" the "teacher" expert system rule by rule firing, but has got only the final conclusion of the "teacher" available at each move. The "student" will try to match the "teacher"'s conclusion by adjusting mutual relations of the interim values of its conclusions on lower levels and its competing conclusions. This way a consistent ranking of "student"'s conclusions is produced, similarly as in phase two. The solution of the corresponding set of inequalities and subsequent training of the neural networks is exactly the same as in phase two.

Certainly, there would be no sense in "training" expert systems using complete expert systems as "teachers". We could use the "teacher" expert systems and not to bother with the "student" expert systems. Thus, the first three phases are preliminaries for the phase four, where a human player is used as a teacher. There is no problem to produce relative ranking of terms (conclusions) at each move throughout the games based on the actual moves by the human player. The problem we shall have to resolve, though, is the fact that unlike "teacher" expert system, the human teacher may be inconsistent (i.e. imperfect teacher) and thus produce an impossible ranking through the many games played.

Because at this stage we have no definite answer how to deal with the inconsistency of human teachers, we have been concentrating mainly on the first three stages in order to establish the viability and practicality of our method of incorporating neural networks in McESE-built expert systems.

2. Uncertainty and McESE

In McESE the outside world is described by n-ary ($n \geq 0$) predicates, which can be negated. Thus, for example, **P** denotes a nullary predicate, **P(x)** denotes an unary predicate, **P(x,y)** denotes a binary predicate and so on. **~P** or **-P** denote the negation of **P**. Predicates are used in McESE as basic statements about facts and objects in the domain in which the corresponding expert system

is supposed to be applied (usually their names are used in mnemonic fashion to indicate the meaning of the predicate): **no_response** is a nullary predicate, and hence a proposition, **tall(peter)** is an unary predicate with its variable instantiated with the "object" **peter**, and thus again a proposition, **loves(peter,mary)** is a binary predicates with its variables instantiated with "objects" **peter** and **mary**, and thus again it is a proposition. When variable(s) of a predicate is (are) instantiated, the predicate becomes a proposition and thus it is valid to ask what is its truth-value, *true* or *false*. In McESE, similarly as in fuzzy logic, a real number between 0 (*false*) and 1 (*true*) is assigned to a predicate with its variable(s) instantiated. This represent the so-called *certainty value* of the predicate for the given instantiation.

Predicates which are never used on the right hand side of any rule in the rule-base are so-called *level 0 predicates*, and they represent basic facts and observations about the given domain. Their certainty values are assigned to them by so-called *service procedures*, which are a part of the expert system. These values reflect subjective "ranking" of facts and observation by the knowledge engineer designing the expert system. Consider, for example, a level 0 unary predicate **high_fever(temperature)**. Its service procedure would return the certainty value for any given value of the temperature. Of course, there is no other reason but the subjective view of the knowledge engineer for assigning .7 to the temperature of 39 degrees centigrade, rather than .8 . In short, the certainty values of the facts and observations as represented by the level 0 predicates are set by the knowledge engineer. *Level 1 predicates* are predicates which occur on the right hand side of some rule where all the predicates on the left hand side are level 0 predicates. They represent conclusions based on facts and observations. *Level 2 predicates* then represent conclusions based on facts and observations and level 1 conclusions, and so on. Predicates on level 1 and up have their certainty values determined by rules.

McESE rules do not use just plain predicates, but so-called *McESE terms* to allow capturing of more complex relations among objects and facts in the domain: a term is an expression of the form

*weight * negation predicate_name (variables) threshold_directive*.

weight is a real number between 0 and 1. It represents the weight of certainty value of the term. When omitted, value of *weight* is assumed to be 1. *negation* is also optional and has value ~ or - , and when used indicates the negation of the subsequent predicate. *predicate_name* is the symbolic name of the predicate and must be present. *(variables)* indicate the arity of the predicate, no variables indicate a nullary predicate, one variable indicates an unary predicate, two variables indicate a binary predicate, and so on. *threshold_directive* is optional and is used to turn the certainty value of the term to boolean values 0 and 1 based on the given threshold. The form of threshold directive is *[op val]*, where *op* is one of the relational operators >, ≤, =, <, and ≥, and *val* is a real value between 0 and 1. The certainty value of an instantiated term (i.e. all variables of its predicate must be instantiated) is determined from the certainty value of its predicate. If the predicate is negated, 1−the value of the (instantiated) predicate is used, otherwise the value of the (instantiated) predicate is used. If this value satisfies the relation given by the threshold directive (if used), the value of the term is changed to 1, otherwise is changed to 0. If the threshold directive is not used, the value is left unchanged. Then the value is multiplied by the weight. This is then the certainty value of the whole term. The McESE rule has the following syntax:

RULEID: T_1 & T_2 & T_3 & ... & T_n =F=> T

where $\mathbf{T}_1$, ... , $\mathbf{T}_n$ and **T** are McESE terms, and **RULEID** is the rule's id, it consists of an identifier followed by a threshold directive, which is optional. When this rule is to be fired, the certainty values of the left hand side terms $\mathbf{v}_1$, ... , $\mathbf{v}_n$ are calculated from the certainty values of corresponding predicates. Then the cvpf **F** is applied to them to produce a certainty value **v**. If the rule's id includes a threshold directive, then the value **v** is compared with it, and if the threshold directive is not satisfied, the rule does not fire. If the threshold directive is satisfied, or if the threshold directive is not used in the rule's id, then the right hand side term **T** is assigned the value **v**. The certainty value of the right hand side (instantiated) predicate is computed from the value of the term **T**. If **T** includes threshold directive, then if the threshold directive is satisfied by **v**, then **v** is changed to 1, otherwise to 0. If threshold directive is not used, **v** remains unchanged. If the right hand predicate is negated, **v** is changed to **1-v**. Finally **v** is multiplied by the corresponding weight. This is the certainty value assigned to the predicate on the right hand side.

McESE-built expert systems allow for both, forward chaining as well as backward chaining. When the knowledge base of such an expert system is queried about a certainty value of a predicate in backward chaining mode, the variables of the predicate must be instantiated, and this instantiation is propagated through the knowledge base down to level 0, and then the evaluation is carried back to the predicate the knowledge base was queried about. In forward chaining mode, variables of a given set of predicates on level 0 must be instantiated, and subsequent evaluation is carried to a specified level. With each predicate a procedure can be associated which is triggered automatically whenever the predicate value satisfies a given condition.

From the brief exposition to McESE we can see that the user of McESE has two mechanisms of dealing with uncertainty: cvpf's and threshold directives. The user does not have to use any, or he may use one, or the other one, or both. These mechanisms are flexible enough that emulation of any of standard techniques for dealing with uncertainty: probabilistic, evidential, fuzzy logic, ad-hoc (like certainty factors of MYCIN), and heuristic, is possible, and in fact, built-in the system (see [J]). Since McESE knowledge base is compiled into data structure resembling a generalized tree, the reasoning with it is in run time reduced to annotation (evaluation) of such a tree, and hence is executed quite fast. The speed of reasoning with McESE rules was also one of the objectives on our mind when we were designing the environment.

Though the software of McESE is entirely written in the programming language C, a form of McESE extension of FranzLISP (see [F]) was used in Expert System Architecture courses (4th year and graduate level courses) for the student projects. Thus a variety of small expert systems using McESE have been built in a few years, ranging from medical diagnostic consultation system for childhood diseases to card game Canasta player. The overwhelming feeling of all students involved had been that it was always more straight forward and simpler to state the declarative part of rules than to specify the corresponding cvpf's. In fact, they usually tried to create a set of rules in boolean form, i.e. to suppress uncertainty involved as much as possible. Even though it ought to be so, we started to thing about the problem of reducing this unpleasant task of the knowledge engineer. In numerous discussions we found that the task of knowledge transfer (i.e. designing a knowledge base) would be much easier if the knowledge engineer were relieved from designing explicit cvpf's.

3. Emulation of CVPF's by neural networks

One of possible solutions to the problem of explicitly stated cvpf's is to use neural networks to replace the cvpf's, and train them for the task. On the one

hand this relieves the knowledge engineer from the drudgery of specifying explicit cvpf's and allows him to concentrate on the relevant high-level relations among objects and facts in the domain and capture them in the rules, but on the other hand it raises the problem of proper morphology of the neural networks used and their proper training.

4. The setup of the project

Two McESE-built expert systems programmed in C to play a card game Canasta are used. The first expert system called "teacher" has a complete rule base (by this we mean a rule base with McESE rules including cvpf's), the second one called "student" is identical, but it lacks cvpf's, which are replaced by neural networks. A program in C called "dealer" facilitates playing of two players, be it two humans, or one human and a program, or two programs. A special program also written in C called "TA" facilitates the "learning" of the "student". Since for every game the "dealer" randomly shuffles the deck of cards, this setup can produce in reasonable time any required number of (significantly) different games, and hence provide a sufficient amount of input data for the training of the "student". This alleviates the problem of preparing reliable training patterns for a number of neural networks. Note that in the canasta player, about a hundred rules and thus about a hundred neural networks are used. The level of success of the training is then verified in a number of games of the "teacher" against the "student".

5. Phase one of the project

While a game between the "teacher" and a copy of the same expert system is played (another option is that a human plays against the "teacher", but it does slow the generation of input data), the "TA" collects the certainty values as produced for each rule firing by the "teacher". Precisely, if a rule $\mathbf{T_1 \& T_2 \& ... \& T_n =F=> T}$ of the "teacher" is fired with values $\mathbf{v_1}$, ... ,$\mathbf{v_n}$ of the left hand side terms, and value **v** is produced by **F** for the right hand side term (i.e. $\mathbf{v = F(v_1, ... ,v_n)}$), then the "TA" stores the feature vector $\mathbf{(v_1, ... v_n)}$ with its desired output **v** in the training set for the neural network associated with the same rule in the "student" knowledge base. When all games are finished, the neural networks are then trained using their respective training sets as created by the "TA".

The main goal of this phase is to determine the viability of the approach, how many games should be sufficient, how the important parameters of neural networks used should be set. The parameters estimated at this stage are: the general morphology of the network (i.e. how many hidden layers), how many nodes in a hidden layer, what value of the gain term (learning rate) should be used, and how strong the error signal should be. As usual, the aim is it to minimize the number of learning sweeps. The back propagation learning algorithms of [RHW] is used.

6. Phase two of the project

As in phase one, the same configuration of the "teacher" expert system, the "student" expert system, and the "dealer" is used. The "TA" program, though, is different. It does not collect the vectors of values, but "builds" a set of inequalities. Let us illustrate the method on a simple example.

Teacher:	Student:
R1: P1 & P2 & P3 =F=> P4	**R1: P1 & P2 & P3 ==> P4**

R2: P5 & P6 =G=> P7 **R2: P5 & P6 ==> P7**
R3: P4 & P7 =H=> P8 **R3: P4 & P7 ==> P8**
R4: P9 & P10 =I=> P8 **R4: P9 & P10 ==> P8**

Thus, **P1**, **P2**, **P3**, **P5**, **P6**, **P9**, and **P10** are level 0 predicates, while **P4** and **P7** are level 1 predicates, and **P8** is a level 2 predicate. Let us call the neural network associated with **R1** of the "student" to emulate **F NF**, the one associated with **R2** of the "student" to emulate **G NG**, and so on.

Let us assume that evaluation of **P8** was requested. The "teacher" first fires the rule **R1**. Let value of **P1** (as returned by its corresponding service procedure as described above; note, both the "teacher" and the "student" have the same service procedure for level 0 predicates) be **.4**, the value of **P2** be **.6**, and value of **P3** be **.8**. The "TA" will store the feature vector **(.4,.6,.8)** and the desired output (symbol) $\mathbf{v}_1$ in the training set of **NF** and remember the value of $\mathbf{v}_1$ (i.e. $\mathbf{v}_1$ **= F(.4,.6,.8)**). Then the "teacher" will fire the rule **R2**. Let value of **P5** be **.8**, and let value of **P6** be **.7**. Then the "TA" will store the feature vector **(.8,.7)** and the desired output (symbol) $\mathbf{v}_2$ in the training set of **NG** and remember the value of $\mathbf{v}_2$ (i.e. $\mathbf{v}_2$ **= G(.8,.7)**). Then the "TA" stores in its permanent storage either $\mathbf{v}_1 < \mathbf{v}_2$, or $\mathbf{v}_1 > \mathbf{v}_2$, or $\mathbf{v}_1 = \mathbf{v}_2$, whatever the case might be and forgets the values of $\mathbf{v}_1$ and $\mathbf{v}_2$. Then the "teacher" fires **R3**. The "TA" will store the feature vector $\mathbf{(v_1,v_2)}$ and the desired output $\mathbf{v}_3$ in the training set of **NH** and remembers the value of $\mathbf{v}_3$**.** Finally, the rule **R4** is fired by the "teacher". Let the value of **P9** be **.9**, and the value of **P10** be **.5**. The "TA" will store the feature vector **(.9,.5)** and the desired output $\mathbf{v}_4$ in the training set of **NI** and remember the value of $\mathbf{v}_4$. Then the "TA" stores in its permanent storage $\mathbf{v}_3 < \mathbf{v}_4$, or $\mathbf{v}_3 > \mathbf{v}_4$, or $\mathbf{v}_3 = \mathbf{v}_4$, whatever the case might be, and forgets the values of $\mathbf{v}_3$ and $\mathbf{v}_4$.

When the training games are over, the training sets of neural networks of the "student" contain feature vectors and corresponding desired output consisting of variables $\mathbf{v}_1$, $\mathbf{v}_2$, and so on, and numbers, while the permanent storage of the "TA" contains a set of inequalities or equalities of the variables $\mathbf{v}_1$, $\mathbf{v}_2$, and so on. These inequalities definitely have a solution, namely the original values of $\mathbf{v}_1$, $\mathbf{v}_2$, ... as produced by the "teacher" and forgotten by the "TA". Linear programming is used by the "TA" to find a solution for the set of inequalities. Then the variables $\mathbf{v}_1$, $\mathbf{v}_2$, ... in the training sets of neural networks of the "student" are replaced by the values of the solution produced by the "TA". Then the training of neural networks follows in the same fashion as in phase one. Note, that because the "TA" forgets the values of the variables $\mathbf{v}_1$, $\mathbf{v}_2$, ... and remembers only the mutual relations of the variables, in fact it only deals with a relative ranking of conclusions of the knowledge base.

7. Phase three of the project

This phase is a modification of phase two. Unlike in the first two stages, the "student" becomes more involved during the training games. The "TA" does not have access to all firings of the "teacher" to produce the feature vectors with desired outputs (to be stored in the training sets of neural networks of the "student") and to construct the set of inequalities (to be solved later). After the "teacher" made its move, the "student" must adjust the mutual relations of variables $\mathbf{v}_1$, $\mathbf{v}_2$, ... (that stand temporarily for the values of conclusions of the knowledge base as described in the previous paragraph) as to arrive to the same conclusion as the "teacher". Thus, the resulting set of inequalities is created by the mutual collaboration of the "TA" and the student. The solution of the set of inequalities, their substitutions in the training sets, and subsequent training of neural networks is as in phase two.

Though the "TA" and the "student" have no access to the rules of the "teacher" and can only "guess" how to arrive to the correct conclusions, the fact that the "student" *reasons* along the same lines as the "teacher" (after all it contains the same rules in its knowledge base) simplifies the task of providing a consistent ranking of conclusions in order to facilitate a solution of the resulting set of inequalities.

8. Phase four, the final stage of the project

As indicated in the introduction, phase four of the project does not have a definite form yet. Phase three was meant as a prerequisite for this phase, for the "TA" and the "student" in that phase are also dealing with a relative ranking of conclusions of the knowledge base. The simplified setup of phase three makes more likely that the set of inequalities as created by the "TA" and the "student" from the ranking of conclusions as produced by the "teacher" has a solution. In phase four, the teacher will be a human, i.e. a game will be played, between two humans, or a human and a program, and the "TA" and the "student" will be "looking over the shoulder" of the human player this time. But, since the "student" will not *reason* in a similar fashion as the human, and since the human may be (and often will be) inconsistent, it may cause the "student" and the "TA" to produce an impossible ranking, i.e. a set of inequalities without a solution. At this stage of our effort, we have not arrived to a satisfactory resolution of this problem.

9. Conclusion

The paper is describing a possible way to incorporate neural networks into McESE-built expert systems to emulate so-called cvpf's (*certainty value propagation function*). The four distinctive stages of the project are described and discussed in their particulars. The preliminary results concerning phase one and partially phase two that have been carried so far by us and graduate students indicate that the approach as described in the paper is viable, and that a three layer perceptron is a suitable configuration for the neural networks for emulation of cvpf's for a particular expert system to play Canasta. Despite the fact that we have not got a satisfactory solution for the problem of inconsistency of human teachers (as discussed in phase four of the project), the method seems interesting and viable enough for us to continue with the project along the stages described here. The rewords of a successful "marriage" of neural networks and expert systems are too enticing.

As a final note, since a McESE knowledge base in its compiled form is a data structure resembling a tree, and since at each node a neural network is "attached" to emulate the cvpf, one can view the whole data structure as a "cascade of neural networks", which leads to the problem of a learning algorithm for such a "cascade of neural networks". If a satisfactory algorithm for training a "cascade of neural networks" were found, the whole project described in this paper would be unnecessary. That would represent a real and significant "fusion" of neural networks and expert systems.

Acknowledgement

This research has been supported by NSERC operating research grants A8034 (I. Bruha) and OGP0025112 (F. Franek).

References

[B] I. Bruha, *Neural Nets: Survey and Application to Waveform Processing*, Intern. Summer Conf. on Art. Intelligence, Dubrovnik, Sept. 1990 (to appear).

[BM] I. Bruha, G.P. Madhavan, *Combined Syntax-Neural Net Method for Pattern Recognition of Evoked Potentials*, in Computing and Information, North-Holland, 1989.

[F] F. Franek, *McESE-FranzLISP: McMaster Expert System Extension of FranzLISP*, in Computing and Information, North-Holland, 1989.

[FB1] F. Franek, I. Bruha, *An environment for extending conven tional programming languages to build Expert System Applications*, Proceedings of IASTED Conference on Expert Systems, Zurich, 1989.

[FB2] F. Franek, I. Bruha, *McESE - McMaster Expert System Environment*, in Computing and Information, North Holland, 1989.

[G] W.A. Gale, *Artificial Intelligence & Statistics*, Addison-Wesley Publ. Comp., 1986.

[H] R. Ho, *A Neural Network System for Recognition of Evoked Potentials*, M.Sc. Thesis, , Dept. of Comp. Sci. & Systems, McMaster University, 1989

[HH] S.J.Henkind, M.C.Harrison, *An Analysis of Four Uncertainty Calculi*, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 18, No. 5, Sept./Oct. 1988

[J] Z. Jaffer, *Different treatments of uncertainty in Expert Systems and their emulation in McESE*, M.Sc. thesis, Dept. of Comp Sci. & Systems, McMaster University, 1990.

[P] J. Pearl, *Probabilistic Reasoning in Intelligent Systems*, Morgan Kaufmann Publ., Palo Alto, 1988.

[RHW] D.E. Rumelhart, G.E. Hinton, R.J. Williams, *Learning Internal Representation by Error Propagation*, in D.E. Rumelhart, J.L. McCelland (Eds.), PDP: Exploration in the Microstructure of Cognition, Vol. 1: Foundations, MIT Press, 1986.

[W] J. Weber, *Principles and Algorithms for Causal Reasoning with uncertainty*, Tech. rep 287, Univ. of Rochester, 1989.