# Simulation of Neural Nets in McESE.

**F. Franek** and **I. Bruha**
Dept. of Comp. Sci. & Systems
McMaster University
Hamilton, Ontario, L8S 4L8 Canada
Email: *franya@mcmaster.ca, bruha@mcmaster.ca*

**Abstract.**

**McESE** - McMaster Expert System Environment - is a software environment to help build expert systems. The production rules in McESE have form $\mathbf{T_1 \ \& \ T_2 \ \& \ ... \ \& \ T_n \ =F=> T}$, where **F** is so-called certainty value propagation function (CVPF), and it takes as its arguments the certainty values of the left-hand side terms $\mathbf{T_1}$, ..., $\mathbf{T_n}$, and returns a certainty value to be assigned to the right-hand side term **T**. In effect, **F** is a procedural knowledge added to the declarative knowledge contained in the rule. The mechanism of CVPFs allows the user of McESE to treat uncertainty in the way he desires. The problems associated with correct specification of CVPFs led the authors to replace CVPFs by neural nets and "train" the resulting hybrid of rule-based expert system and neural nets. It was realized during the project that the ability to specify these neural nets from within the McESE system was crucial in order to carry out the needed tests and experiments. It turned out that the machinery of McESE (i.e. rule parser, rule compiler, and inference engine) after slight enhancements allows to simulate neural nets within McESE with any topology desired. In the paper these enhancements and their background are discussed.

## 1. Introduction.

It is well known that both major directions of AI research into intelligent systems Neural Nets and Expert Systems are almost complementary in their strengths and weaknesses; while Neural Nets are not so good in higher-level reasoning (mainly for the lack of symbolic representation and explanation facilities), they are reasonably good in imprecise classification and recognition; on the other hand Expert Systems are reasonably good in higher-level reasoning, but they are fundamentally weak in handling imprecise and uncertain knowledge. This weakness in handling uncertainty in Expert Systems has been addressed by many researchers. The authors of this paper have also contributed a bit towards the solution of the problem. Our approach, though, was more practical than theoretical

(mainly because most of the main methods of dealing with uncertainty in Expert Systems are rigorous on their surface only, in their essence they are more ad-hoc methods than well-founded theories - for a more thorough discussion see e.g. [J]). We have designed and built a software tool to help create expert systems, so-called **McESE** - *McMaster Expert System Environment* (see [FB1], [FB2], [F], [L], [C]). We had had several objectives on our mind when we were designing the system, but one of them was to give the user of McESE building an expert system a possibility to deal with uncertainty in any way he desires.

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

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

where $\mathbf{T_1}$, ... , $\mathbf{T_n}$, and **T** are so-called **McESE terms**. A McESE term consists of a **weight** (which has the same range as McESE certainty values, i.e. real numbers between 0 and 1 inclusive) -when omitted the default value is 1, followed by a **predicate** (with or without variables), and possibly followed by a **threshold directive**. The predicate can be negated. The certainty value of such a term is determined from the certainty value of the predicate for the given instantiation of variables. The certainty value of a negated predicate is 1 less the certainty value of the predicate. If the threshold directive is used, then (a) if the certainty value of the predicate is bigger than (or bigger or equal to, it depends on the form of the threshold directive) the value given by the threshold directive, the certainty value of the term is 1, else (b) the certainty value of the term is 0. In case the threshold directive is omitted, the certainty value of the predicate becomes the value of the term. As the final step, this certainty value is multiplied by the weight of the term.

The so-called **CVPF** (*certainty value propagation function*) **F** takes as its arguments the certainty values of the left-hand side terms $\mathbf{T_1}$, ... , $\mathbf{T_n}$ and returns the certainty value to be assigned to the right-hand side term **T**. Thus, each rule contains not only the declarative knowledge (the relations among different predicates and their weights), but also a procedural knowledge (**F**) of how to handle and propagate uncertainty within this particular rule. **F** can be a user-created program, or any of the McESE built-in functions (see [J]). Each rule can have a different CVPF, or several rules can share one. It is clear that CVPFs ought to be non-decreasing functions: when the certainty of one of the left-hand side terms increases, the certainty of the right-hand side term should not decrease.

The basic McESE software (for all extensions), i.e. rule parser, rule compiler, and the inference engine together with the explanation component are programmed in C. The reason for the choice of C was threefold: the speed of execution (and hence the speed of McESE inference), the compactness of the resulting code, and the low-level functionality of C allowing to work with complex data structures.

While experimenting with McESE and building sample expert systems we had realized that it had always been easier to state a rule than to give the corresponding CVPF. So we proposed to "fuse" Expert Systems and Neural Nets to emulate the CVPFs (see [FB3]). Such combination separates and preserves the strong aspects of both methods, and complements them in their weak aspects.

One of the important goals in the design of McESE was to allow the user to build expert systems using the "usual" programming methods and languages. This is facilitated by "extending" a particular programming language so it can deal with McESE knowledge bases. At this point of time extensions of LISP (so-called McESE-FranzLISP, see [F], and McESE-Allegro CL, see [C]), an extension of SCHEME (so-called McESE-SCHEME, see [L]), and an extension of C (so-called McESE-C) are completed. In a particular McESE extension the whole expert system is thus built using just one programming language (that one of the extension) plus the language of McESE rules. This "homogeneity" of McESE extensions had caused some problems to our project, for it required that the neural nets for the emulation of CVPFs be programmed outside of McESE and carried as an "appendix" to a McESE extension.

It was realized early in the project that the ability to specify these neural nets from within the McESE extension was crucial in order to carry out the needed tests and experiments. It turned out that the McESE rule formalism together with some custom made CVPFs and the way the McESE inference engine works allow us to define and built within any McESE extension (be it McESE-FranzLISP, McESE-Allegro CL, McESE-SCHEME, or McESE-C) an expert system that simulates desired behaviour of a neural net with a particular topology. Since every McESE extension allows that a CVPF is another expert system, the expert systems simulating CVPFs can easily be integrated with the underlying expert system into a single expert system. Thus the whole project of emulation of CVPFs as described above can be carried within a McESE extension. This simplifies the project significantly, for it allows of testing of different expert systems programmed in different

languages, as well as experimenting with the neural nets during the testing.

**2. How the McESE inference works.**

In order to describe the simulation of neural nets in McESE, we have to discuss the working of the McESE inference engine in greater details.

A backward chaining inference is triggered by a query of a knowledge base, i.e. a request to evaluate a particular predicate. Of course, in the query the variables of the predicate must be instantiated with programming "objects". The inference engine will evaluate all rules in the knowledge base whose right-hand side term contains the given predicate. Since there may be more than one such rule, only the maximal and minimal evaluations are recorded and the maximal (or minimal, it depends on the form of the query) evaluation is returned to the expert system by the inference engine. Only the rule whose CVPF returns a value that exceeds a threshold given in the rule (if any) is considered "fired". The evaluation of rules is recursive: in order to evaluate a rule, all its left-hand side terms must be evaluated (by a recursive call to the inference engine), of course with all variables of the left-hand side predicates instantiated with the "objects"; when completed, the corresponding CVPF is applied to the certainty values of the left-hand side terms and the resulting value is assigned to the right-hand side term, from which the value of the predicate is determined.

For example: consider a simple rule

**R1[>.3]: 0.8*P(x)[>=.5] & Q(y) =F=> ~R(x,y).**

The query is to evaluate **$R(O_1,O_2)$**, where **$O_1$** and **$O_2$** are some programming objects (functions, variables, structures, pointers, ... ). Firstly, the inference engine is called to evaluate **$P(O_1)$**. Let us assume that the inference engine returns 0.7. The value of the term **P(x) [>=.5]** will be set to 1, as 0.7 >= 0.5 (**[>=.5]** is the threshold directive of the first term). Thus the value of the term **0.8*P(x) [>=.5]** is set to 0.8. Secondly, the inference engine is called to evaluate **$Q(O_2)$**. Let us assume that the inference engine returns 0.5. Hence the value of the term **Q(y)** is set to 0.5. Thirdly, the inference engine invokes the CVPF **F** with arguments 0.8 and 0.5. Let us suppose that **F** returns 0.4. Hence the value assigned to the right-hand side term **~R(x,y)** is 0.4, and so the value of the predicate **$R(O_1,O_2)$** is set to the

value (1-0.4), i.e. 0.6. Since 0.4 > 0.3 (as stipulated by the threshold directive of the whole rule), the rule is considered "fired" and the value of $\mathbf{R(O_1,O_2)}$ is left as 0.6.

It is clear that this recursive (backward chaining) calling of inference engine must stop somewhere. In fact it does stop when so-called **level 0 predicates** are reached: those are predicates which occur only on left-hand side of rules (at this point it is important to remark that no reasoning cycles are allowed in McESE rules, so all predicates are stratified into levels). The interpretation of level 0 predicates is that they represent facts and observations, while predicates on higher levels represent conclusions based on facts and observations, and conclusions of lower levels. The level 0 predicates are evaluated by so-called **predicate service procedures**, which represent the connection of the knowledge base to the "real" world - they evaluate the level 0 predicates based on the input data.

If a rule has no CVPF specified, then the **default CVPF** is used during the inference. The user of McESE can specify the default CVPF. It can be any of the McESE built-in functions, or a user-programmed one.

From the above description of the way the inference engine works, it is clear that during inference evaluation of a predicate several other predicates may have been evaluated as well. A McESE function **eval** can fetch this value for any particular predicate (which has been evaluated during the last call to inference engine from the top level - this constitutes the so-called **inference cycle**).

**3. The compiled McESE knowledge base.**

When the inference engine is evaluating a rule and is about to apply the corresponding CVPF, it finds the address of that subprogram (which realizes the CVPF) in the so-called **compiled knowledge base**, which is a stratified linked data structure resident in main memory. The McESE compiler, when parsing the rules and building the corresponding data structure, prepares a list of all CVPFs used in the knowledge base. When the knowledge base is opened (and loaded into main memory), the addresses of corresponding subprograms are filled in, including the built-in functions. If a CVPF is one of the built-in functions, the user of McESE does not have to specify it (i.e. program it), he just refers to it by name in a rule. This setup allows us to extend the set of built-in functions as we desire without a need to modify McESE in any significant fashion. It is also clear that

from the programming point of view an expert system can be used as a CVPF (the expert system being nothing else but a program), as long as it has an appropriate input (certainty values) and an appropriate output (a certainty value). (Not only the CVPFs can be experts systems, also a predicate service procedure can be implemented as an expert system. This allows for a natural hierarchical integration of McESE expert systems, but this topic exceeds the scope of this paper and so we shall not discuss it here.)

In the compiled knowledge base each rule has a "node" in which each term has its own memory, where its weight, sign, and threshold directive are "remembered", and where a link to the corresponding predicate "node" is stored. In each predicate node the last minimal and maximal evaluations are stored, together with the last instantiation of the variables.

**4. The methodology of the simulation of Neural Nets in McESE.**

The topology of the multilayer perceptron to be simulated is given by the McESE rules, where the terms on the left-hand side represent nodes in one layer, and the term on the right-hand side represents a node in the next layer. In this way the size of each layer and the number of layers are defined.

For example the simple net with one hidden layer on Fig.1 can be given by rules:

$\mathbf{R_1: I_1 \ \& \ I_2 \ \& \ I_3 \ \& \ I_4 => H_1}$
$\mathbf{R_2: I_1 \ \& \ I_2 \ \& \ I_3 \ \& \ I_4 => H_2}$
$\mathbf{R_3: H_1 \ \& \ H_2 => O_1}$
$\mathbf{R_4: H_1 \ \& \ H_2 => O_2}$

Thus, in general, the input nodes would be represented by level 0 McESE predicates, the first hidden layer nodes by level 1 McESE predicates, and so on. Since all McESE terms can be weighted (an omitted weight is 1 by default), we can preset all the weights of the simulated neural net in the rules. Rules

$\mathbf{R_1: w_{1,1}*I_1 \ \& \ w_{2,1}*I_2 \ \& \ w_{3,1}*I_3 \ \& \ w_{4,1}*I_4 => H_1}$
$\mathbf{R_2: w_{1,2}*I_1 \ \& \ w_{2,2}*I_2 \ \& \ w_{3,2}*I_3 \ \& \ w_{4,2}*I_4 => H_2}$
$\mathbf{R_3: w_{5,1}*H_1 \ \& \ w_{6,1}*H_2 => O_1}$
$\mathbf{R_4: w_{5,2}*H_2 \ \& \ w_{6,2}*H_2 => O_2}$

will set the weight of the connection from $\mathbf{I_1}$ to $\mathbf{H_1}$ to $\mathbf{w_{1,1}}$, the weight of the connection from $\mathbf{I_1}$ to $\mathbf{H_2}$ to $\mathbf{w_{1,2}}$, and so on.

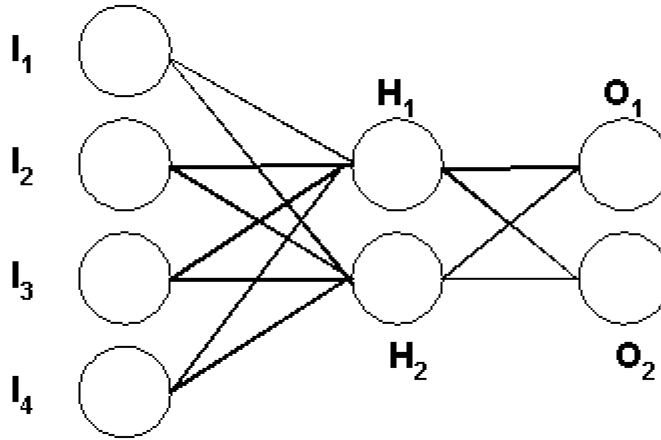


Fig. 1

Instead of pre-setting the wiegtht by hand, the McESE built-in function **randweights** can be used to set the weights in a random fashion by specifying an extra rule whose left-hand side terms are the output nodes, and the right-hand side term is a spurious term **RAND**, and the CVPF is the McESE built-in function **randweights**, i.e.

**$R_5$: $O_1$ & $O_2$ =randweights=> RAND**.

During the initialization of the neural net the inference evaluation (in backward chaining mode) of **RAND** will cause the random setting of all weights in the neural net.

The default CVPF for the knowledge base representing the neural net is set to **sigma**, which is one of the McESE built-in functions and which represents the so-called nonlinearity (the sigmoid function) of the net.

To simulate the classification mode of the multilayer perceptron, all the output nodes (i.e. $\mathbf{O_1}$ and $\mathbf{O_2}$) must be evaluated given input values (i.e. $\mathbf{i_1}$, $\mathbf{i_2}$, $\mathbf{i_3}$, and $\mathbf{i_4}$). To do so, and extra rule is added:

**$R_6$: $O_1$ & $O_2$ =nil=> CLASSIF**

with a spurious right-hand side term **CLASSIF**. Note that the McESE built-in function **nil** returns 0 to any input. The inference evaluation of the spurious node **CLASSIF** is requested. The inference engine in its backward chaining mode will evaluate all output nodes, thus the inference evaluation of $\mathbf{O_1}$ is requested first. The inference engine will evaluate all nodes in the hidden layer, $\mathbf{H_1}$ and $\mathbf{H_2}$, and to do so it must first evaluate the level 0 predicates, i.e. nodes $\mathbf{I_1}$, $\mathbf{I_2}$, $\mathbf{I_3}$, and $\mathbf{I_4}$. The predicate service procedures for $\mathbf{I_1}$, $\mathbf{I_2}$, $\mathbf{I_3}$, and $\mathbf{I_4}$ must be set up to fetch the (normalized) input values $\mathbf{i_1}$, $\mathbf{i_2}$, $\mathbf{i_3}$, and $\mathbf{i_4}$ (interpreted as certainty values of the corresponding terms $\mathbf{I_1}$, $\mathbf{I_2}$, $\mathbf{I_3}$, and $\mathbf{I_4}$). Then rules $\mathbf{R_1}$ and $\mathbf{R_2}$ are fired to evaluate nodes $\mathbf{H_1}$ and $\mathbf{H_2}$. For example the left-hand side terms of rule $\mathbf{R_1}$ will be evaluated to $\mathbf{w_{1,1}*i_1}$, $\mathbf{w_{2,1}*i_2}$, $\mathbf{w_{3,1}*i_3}$, and $\mathbf{w_{4,1}*i_4}$, and so, using the default CVPF **sigma**, $\mathbf{H_1}$ will be assigned the value $\mathbf{h_1} = \mathbf{sigma(w_{1,1}*i_1,w_{2,1}*i_2,w_{3,1}*i_3,w_{4,1}*i_4)}$. Similarly, $\mathbf{H_2}$ will be assigned the value $\mathbf{h_2} = \mathbf{sigma(w_{1,2}*i_1,w_{2,2}*i_2,w_{3,2}*i_3,w_{4,2}*i_4)}$. Now, the value of $\mathbf{O_1}$ and can be determined. The rule $\mathbf{R_3}$ is fired by the inference engine and **O1** is assigned the value $\mathbf{o_1} = \mathbf{sigma(w_{5,1}*h_1,w_{6,1}*h_2)}$. Then, the inference value of $\mathbf{O_2}$ is requested and since now the values of $\mathbf{H_1}$ and $\mathbf{H_2}$ are known to the inference engine (as it is within the same inference cycle), rule $\mathbf{R_4}$ is immediately fired producing the value $\mathbf{o_2} = \mathbf{sigma(w_{5,2}*h_2,w_{6,2}*h_2)}$ for $\mathbf{O_2}$. Now the evaluation of **CLASSIF** can be completed by firing the rule $\mathbf{R_6}$ and assigning the value 0 to **CLASSIF**. The McESE function **eval** can be now used to fetch the values of $\mathbf{O_1}$ and $\mathbf{O_2}$, producing the complete vector of output values.

To simulate the learning mode of the neural net, an extra rule is needed: its left-hand side terms are the output nodes, and the right-hand side term is a spurious term **LEARN**, and the CVPF is the McESE built-in function **backprop**, i.e.

$\mathbf{R_7}$**:** $\mathbf{O_1}$ **&** $\mathbf{O_2}$ **=backprop=> LEARN**.

The whole process starts with a request for inference evaluation of **LEARN**. The whole process up to the evaluation of all output nodes (i.e. $\mathbf{O_1}$ and $\mathbf{O_2}$) is the same as for the classification mode, with the exception that the input values (i.e. $\mathbf{i_1}$, $\mathbf{i_2}$, $\mathbf{i_3}$, and $\mathbf{i_4}$) are taken from the training set, again through the predicate service procedures. Then rule $\mathbf{R_7}$ is fired, and the inference engine applies the CVPF **backprop** to the values $\mathbf{o_1}$ and $\mathbf{o_2}$. The McESE built-in CVPF **backprop** is a subprogram to realize the *back propagation training algorithm*. The reader can refer for a detailed description of this algorithm to [B], [BH], or [BMC]. For the purpose of this paper it suffices to say that **backprop** compares the vector of

expected output values from the training set with the vector of actual output values (i.e. **[$o_1$,$o_2$]** for our example) obtained through the inference, and according to parameters *maximum value for error signal*, *learning rate*, and *momentum term*, modifies the weights throughout the whole knowledge base, if necessary. **backprop** evaluates **LEARN** to 0 if there has been a modification of a weight, if no modification has occurred, **LEARN** is evaluated to 1. This is necessary for the user to be able to recognize at the top level when the learning process stopped.

**5. Conclusion.**

The advantage of having the machinery of McESE take care of all aspects of the specification and the behaviour of neural nets required in our project significantly outweighed the effort needed to enhance the McESE. The set of McESE built-in functions had to be extended by CVPFs **randweights** (to randomize weights), **sigma** (to define the nonlinearity of neural nets), and **backprop** (to realize the back propagation learning algorithm). The functions **randweights** and **backprop** must be built-in, for they require a low-level access to the compiled knowledge base representing the net (i.e. to access to the weights), which is not possible for the user from the top level. On the other hand, the function **sigma** can be programmed at the top level, and hence replaced by the user, if he desires so. The addition of **randweights** and **backprop** to the set of McESE built-in functions did not pose any problems, thus the programming effort was all what was required.

In the future, if need be, other training algorithms may be added in the form of McESE built-in functions. Meanwhile, the research of ours into a training algorithm for a "cascade" of neural nets (and that what a compiled McESE knowledge base with neural nets to emulate CVPFs in fact is) will continue. Such an algorithm would represent a significant contribution towards a "seamless" fusion of Expert Systems and Neural Nets.

Bibliography

[B] I. Bruha, *Neural Nets: Survey and Application to Waveform Processing*, Informatica, 1 (1991), 27-42

[BH] I. Bruha and R. Ho, *Evoked Potential Waveform Processing By a Two-Layer Perceptron: Heuristics for Optimal Adjustment of Its Parameters*, Neuronet'90,

Prague (1990), 285-287

[BMC] I. Bruha, G.P. Madhavan, and M.S-K. Chong, *Use of Multilayer Perceptron for Recognition of Evoked Potentials*, Internl. J. Pattern Recognition and Artificial Intelligence, 4, 4 (1990), 705-716

[C] A. Chlobowski, *McESE Extension of Allegro Common Lisp*, M.Sc. thesis, Dept. of Comp. Sci. & Systems, McMaster University, 1991.

[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 conventional 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.

[FB3] F. Franek, I. Bruha, *A Way to Incorporate Neural Networks into Expert Systems*, Proceedings of IASTED Conference on Artificial Intelligence Application & Neural Networks, Zurich, 1990.

[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.

[L] D. Lentz, *Notes on the Implementation of McESE SCHEME*, Technical report 90-01, Dept. of Comp. Sci. & Systems, McMaster University, 1990.