TM-0472

GHCによる実験的ネットワークプログラミング

奥村　晃

March, 1988

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

ＧＨＣによる実験的ネットワークプログラミング
（暫定版：昭和６２年２月７日）

奥村 晃

(財)新世代コンピュータ技術開発機構
〒１０８ 港区三田１丁目４番２８号三田国際ビル２１階

概要

並列論理型言語ＧＨＣは非常に粒度の細かい並列環境を提供する。ＧＨＣを用いて並列に動作する多数の永続的プロセスによる計算モデルを構築するのは容易であり、コネクショニストモデル等で用いられるネットワーク表現を実験的に扱うのに適していると思われる。本稿では、ＧＨＣを用いてネットワークを表現することの優位性を主張し、その実現例を示す。また、記述をさらに容易にするためのシンタクスシュガーの提案を行う。

１．はじめに

近年、コネクショニストモデルあるいはＰＤＰモデルと呼ばれるようなニューラルネットワークによるパターン認識、知識処理の研究が進められ、その有用性が注目されている。ネットワーク表現は活性状態を保持するいくつかのユニットとユニット間を結合する重み付リンクから成り、活性度の伝播によって処理を行う。またエラーの逆伝播に基づいてリンクの重みを変更することによって学習を可能にしている[Ru86]。

ネットワーク表現の有効性は NETTALK、舟の操舵等のいくつかの適用事例で明らかにされている。しかしながら、単純な処理でも比較的多数のユニットを必要とするため計算量の問題が生じる。入力ユニット、出力ユニットの他に隠れユニットと呼ばれる中間層や、また自然言語処理等で様々な概念や中間カテゴリを表現するためのユニット等、様々なユニットが必要となり、またリンク数はユニット数に輪をかけて増大する。また、ネットワークが平衡するまでには数多くの伝播処理が必要である。逆伝播による学習では簡単なものでも数千回の試行を必要としたりする。

そのため理想的には専用のハードウエアが必要で、ニューロコンピュータ等と呼ばれて研究がされている。しかし、現時点ではこれらは一般に入手可能なものではなく、数多くの入門研究者が気軽に実験に使用するという訳にいかない。そのため汎用機やワークステーションでのシミュレーションが現実的である。

本稿は、ＩＣＯＴで提案された並列論理型言語ＧＨＣ[Ue85]によるニューラルネットのシミュレーションを提唱するものである。ＧＨＣでは非常に粒度の細かい記述が可能

で、並列に動作する多くのプロセスからなる処理モデルを簡単に構築することができる。また、部分的にコンパイルし直すことやプログラムを追加することが容易なため、モデルを適宜修正するような実験的ネットワーク構築に向いていると思われる。

また、ＧＨＣは基本的には人工知能向き言語Prologを並列化したものであり、知的処理を行うのに充分な記述力を持っている。そのため人間の持つ知識のうち規則化が容易なものについてはＧＨＣプログラムとして直接記述し、そうでない部分をネットワークによって表現する等の柔軟なシステム構成が考えられる。例えば自然言語処理において構文解析部分を論理文法によって実現し、意味処理をネットワークに分担させる事によって両者を融合することができる。岡のモデル[岡87]もこの種類のモデル構築例である。

２章でＧＨＣによる永続的プロセスの表現、３章では永続的プロセスによるネットワークの構築方法、４章では逆伝播アルゴリズムの実現について説明する。ここまではＧＨＣによる言語の実現であるが、５章ではネットワーク表現を容易にするためのシンタクスシュガーの導入を行い記述例を紹介する。ＧＨＣでの実現に当たっての問題点を６章で、今後の展望について７章で述べる。

２．永続的プロセスの記述

ＧＨＣの永続的プロセスは述語の再帰呼出しによって実現される。停止条件Ｃが満たされるまで手続きＡを繰返し実行するプロセスＰは次のような記述形態になる。

```
P :- <Cが偽> | A, P
P :- <Cが真> | <終了処理>
```

ＧＨＣではゴールはすべてＡＮＤ並列で実行されるため、こういったプロセスをいくつも生成することが簡単にできる。Ｐのプロセスと Ｑのプロセスを同時に起動したければ、(Ｐ，Ｑ)と並べて書くだけでいい。またＰとＱは共有変数を通じてメッセージ交換ができる。(p(X),q(X))というゴールが起動されると、一方がＸに書き込んだ内容を他方が読み出すということができる。共有変数による通信は通常ストリームという構造データを用いて行う。ストリームはPrologのリスト構造で表現されるが、ＧＨＣでは生成側が構造の一部を決定しただけで消費側が逐次読み出すことができる。構造の残りの部分が未定の段階から処理が進められることと、処理の終わった部分がどんどん捨てられてしまうところからストリームと呼ばれる。

ネットワーク表現に現れるユニットやリンクは、入力された数値になんらかの計算を施して出力を出すことを繰返し行う。これを入出力をストリームで行う永続的プロセスで表現する。例えば入力ストリームの各要素の数値を２倍にして出力するプログラムtwiceは次のようになる。

```
twice(In,Out) :- In=[N|Ns] | X:=N*2, Out=[X|Xs], twice(Ns,Xs).
twice(In,Out) :- In=[]     | Out=[].
```

3．ネットワーク表現

ＧＨＣでニューラルネットを表現するにはリンクやユニットに対応する永続的プロセスを生成しストリームで結合すればいい。リンクのプロセスは受け取った入力とリンクの重みとの積を求めて出力する。ユニットは入力を合計して域値を超えていれば１を出力するし、そうでなければ０を出力することにする。

図１にニューラルネットの例を示す。

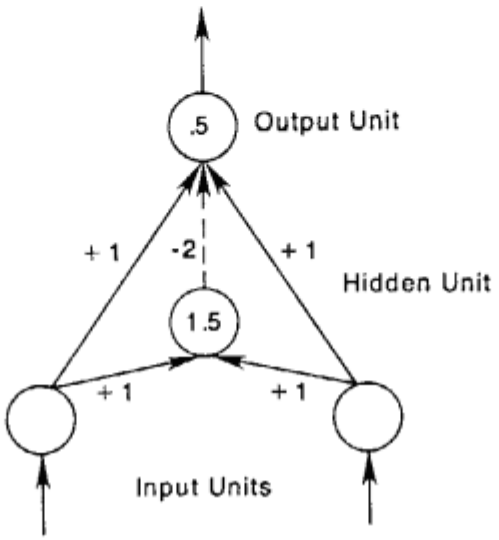

図１　排他的論理和を求めるネットワーク（[Ru86]より）

これは２つの入力の排他的論理和を求めて出力するものである。円はユニットで円中の数値は域値である。リンクの側の数値はリンクの重みである。－２の重みを持つリンクが破線で表されているがこれは抑制性のリンクであることを示している。ここで入力ユニット（Input Units）とは外部からの入力の各々に対応して１つづつ存在するもので、入力をそのまま出力するだけなのでＧＨＣのプロセスとしては特に生成する必要はない。隠れユニット（Hidden Unit）と出力ユニット（Output Unit）は、直接外部に接続していないか外部に出力するかの違いだけで同じ振舞を要求される。そのためプログラム上、両者は同様のプロセスで表現される。

プログラム１は図１のネットワークをＧＨＣで表現したものである。

```
xorNet(In1,In2,Out) :- true |
        link(In1,InH1,1), link(In2,InH2,1),
        hiddenUnit(InH1/0,InH2/0,OutH),
        link(In1,InO1,1), link(In2,InO2,1), link(OutH,InO3,-2),
        outputUnit(InO1/0,InO2/0,InO3/0,Out).

link([In1|Ins],Out,Weight) :- true |
        Out1:=In1*Weight, Out=[Out1|Outs],
        link(Ins,Outs,Weight).
link([],Out,_) :- true | Out=[].

hiddenUnit([A1|A]/_,B/B1,Out) :- true |
        Sum:=A1+B1, greater(Sum,1.5,Out1), Out=[Out1|Outs],
        hiddenUnit(A/A1,B/B1,Outs).
hiddenUnit(A/A1,[B1|B]/_,Out) :- true |
        Sum:=A1+B1, greater(Sum,1.5,Out1), Out=[Out1|Outs],
        hiddenUnit(A/A1,B/B1,Outs).
hiddenUnit([]/_,[]/_,Out) :- true | Out=[].

outputUnit([A1|A]/_,B/B1,C/C1,Out) :- true |
        Sum:=A1+B1+C1, greater(Sum,0.5,Out1), Out=[Out1|Outs],
        outputUnit(A/A1,B/B1,C/C1,Outs).
outputUnit(A/A1,[B1|B]/_,C/C1,Out) :- true |
        Sum:=A1+B1+C1, greater(Sum,0.5,Out1), Out=[Out1|Outs],
        outputUnit(A/A1,B/B1,C/C1,Outs).
outputUnit(A/A1,B/B1,[C1|C]/_,Out) :- true |
        Sum:=A1+B1+C1, greater(Sum,0.5,Out1), Out=[Out1|Outs],
        outputUnit(A/A1,B/B1,C/C1,Outs).
outputUnit([]/_,[]/_,[]/_,Out) :- true | Out=[].

greater(X,Y,Out) :- X>Y | Out=1.
greater(X,Y,Out) :- X=<Y | Out=0.
```

プログラム１　ＧＨＣによる排他的論理和ネットワーク

トップレベルはxorNet/3で、２入力１出力である。xorNet/3は５つのリンクプロセスと各１つの隠れユニットプロセスと出力ユニットプロセスを生成する。計７つのプロセスはストリームで結合される。

リンクプロセス（link/3）は入力ストリーム（第一引数）に要素が到着したらそれにWeightを掛けて出力ストリーム（第二引数）に流す。

隠れユニットと出力ユニットのプログラムは入力の数が違うだけでほとんど同じである。入力引数は斜線（slach）で結合された構造で斜線の左は入力ストリーム、右は最近の入力値を保存する。いくつかある入力ストリームの何れかに要素が到着すると、まず入力の合計が計算される。要素が到着していない入力については最近の数値を用いる。入力の合計が域値を超えているかどうかによって出力を決定するのがgreater/3である。その結果を出力ストリームに流して自分を再帰的に呼び出す。

プログラム１の実行結果を図２に示す。

```
| ?- ghc xorNet([0,0,1,1,1,1,0,0,0,0],[0,0,0,0,1,1,1,1,0,0],Out).
720 msec.

Out = [0,0,1,1,1,1,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0]
```

図２ プログラム１の実行結果

４．逆伝播アルゴリズム

逆伝播アルゴリズム（back-propagation algorithm）はネットワークの出力と理想値との差をエラーとして出力ユニットから入力ユニットに向けて逆伝播して行き、エラー量に応じてリンクの重みを調節するものである。

これの簡単な実現方法としては、各ストリームを双方向に張り通常の入出力のほかにエラー量の入出力ストリームを用意することが考えられる。プロセスはまず入力を待って出力を行い、次にエラーの到着を待ってさらにエラーを伝播する。しかしこれではプロセスに二つの状態が必要ですっきりしない。

そこでエラーの伝播を行うために未完成メッセージ（incomplete messages）を用いる。未完成メッセージとはストリームの要素に不定項を組み合わせて流す方法である。例えば、いま数値１を出力するものとする。このとき変数Errorを組み合わせて(1.Error)という構造をストリームに流せばこれを受け取った側はエラー量を変数Errorに結合して送り側に逆送する事ができる。

付録１に逆伝播アルゴリズムを実現したプログラム例を示す。これは実験に用いたもので端末入出力プロセスが付属しており、また隠れユニットと出力ユニットの数をトップレベルで指定できる等、幾分汎用性を持ったものである。プログラムの説明は省略する。

５．シンタクスシュガーの導入

ネットワークを構成するプロセス群の定義は、どれも末尾再帰のストリーム入出力を行うものである。そのため当然酷似した定義がプログラム中に多数現れることになる。そこで類型化している定義の記述の便を図るためにシンタクスシュガーの導入を考える。まず永続的プロセスの記述にはホーン節を用いることにする。このホーン節は入力と出力の関係を一つ一つの要素レベルで扱い、引数はストリームの形状を取らない。このホーン節をＧＨＣに変換した時点で引数をストリーム化する。そのため引数が入力か出力かを明示する必要がある。このため後置オペレータ?と^を導入する。X?は入力引数、Y^は出力引数である。例えば２入力の和を出力するプロセスの定義は次の様になる。

```
sum(X?,Y?,Out) :- Out is X*Y.
```

これはコンパイラの前変換で次のＧＨＣプログラムになる。

```
sum([X1|X],[Y1|Y],Z) :- true | Z1:=X1*Y1, Z=[Z1|Zs], sum(X/X1,Y/Y1,Zs).
sum([X1|X]/_,Y/Y1,Z) :- true | Z1:=X1*Y1, Z=[Z1|Zs], sum(X/X1,Y/Y1,Zs).
sum(X/X1,[Y1|Y]/_,Z) :- true | Z1:=X1*Y1, Z=[Z1|Zs], sum(X/X1,Y/Y1,Zs).
sum([]/_,[]/_,Z) :- true | Z=[].
```

１行目は最初の入力用で、両方の入力が揃うのを待っている。それ以降ではどちらかの新しい入力によって計算が進められ新しい入力が届いていないほうの引数については保存してある最新の数値（斜線の後ろ）を用いる。

このシンタクスシュガーを用いて逆伝播アルゴリズムを実現したプログラムを付録２に示す。ネットワークは２入力１出力で隠れユニットが３つある。各入力ユニットと各隠れユニットの間、および各隠れユニットと出力ユニットの間には須らくリンクが存在し、入力ユニットと出力ユニットを直接結合するリンクは存在しない。

シンタクスシュガーを用いて画一的に記述する事を目指し、未完成メッセージ等の複雑な構造の使用や一つのプロセスによる双方向の入出力の処理を避けて実現した。そのため、逆伝播の処理は別系統のプロセス群が処理することになり、リンクプロセスは新しい重みをストリームで受け取る形になっている。

６．問題点

４．で示した逆伝播アルゴリズムについてＰＳＩマシン上のＧＨＣコンパイラを用いて排他的論理和の学習実験を行ったがメモリ不足で失敗に終わった。目標としては数千回の試行回数で評価を行いたかったのだが百数十回がせいぜいであった。

原因としてはプログラムに余分な機能があって最適化されていないということや、ＰＳＩ上の処理系が汎用機上のものをそのまま移植したもので最適でないということも考

えられるが、それよりも数千回の試行というのが莫大な計算量を必要とする点にあると考えられる。

計算量の問題については、将来実用的なＧＨＣ処理系の実現やＰＩＭのような専用マシンによる速度向上が予想される。また、ＧＨＣプログラムの変換技法による最適化も考えられる。現段階では本質的に必要な計算量を算定し、将来期待できる計算パワーの基で何れ程の規模の実験が可能であるかの予想をしておくべきだと思われる。

7．将来の展望

ＧＨＣによるニューラルネットの手軽な実験方法について述べた。逆伝播による学習を行うには現時点では計算量の面で難があると思われるが、様々な静的モデルを構築して実験を行うには適当だと考える。

シンタクスシュガーについてはさらに検討が必要である。現段階のものはリンクやユニットのプロセスの定義部分を定式化したのみであるが、プロセスを結合する部分はＧＨＣそのものである。この結合部分についても定式化を行いネットワーク記述言語を提案したい。これはＧＨＣの上位言語の一つとして捉えることができるかもしれない。また、定式化によって一つのプロセスの機能が制限されているが、この枠内でどの様な計算が可能であるかということも評価したい。もしもそれで記述力が充分ならネットワークプログラミングというプログラミングパラダイムとしてさらに検討を進めたい。制限することによって効率的な実現が考えられるであろうから、並列マシンの機械語として考えることができるかもしれない。

実験としてはニューラルネットの範囲に止まらずリンクによって記号が伝達されるような高次のネットワークを考えたい。岡のモデルについても実際的な例題について実験を行いたい。

文献

[岡87] 岡夏樹，意識処理／無意識処理の認知モデル，日本ソフトウェア科学会第４回大会論文集，1987，pp.459-462.

[Ru86] Rumelhart, D.E. 他, Learning Internal Representations by Error Propagation, PARALLEL DISTRIBUTED PROCESSING 第一巻, The MIT Press, 1986, pp.318-362.

[Ue85] Ueda, K., Guraded Horn Clauses, ICOT Technical Report TR-103.

付録1　逆伝播アルゴリズムの実現例

```
go(N1,N2) :- true |
        instream(InSeq),
        prepareIO(N2,InSeq,start,Input,start,Target,[start|Outstream]),
        model(N1,N2,Input,Output,Target),
        constructOutput(Output,Outstream).

prepareIO(N,Seq,LastI,Input,LastT,Target,[O|Os]) :- true |
        Seq = [nl,
                write('==> '), write(O), nl, nl| Seq1],
        difference(LastT,0,0,TotalDif),
        Dif := TotalDif / N,
        prep0(N,Seq1,LastI,Input,LastT,Target,Dif,Os).

difference(_,start,_,Dif) :- true | Dif = 0.
difference([T|Ts],[O|Os],D0,D) :- true |
        D1 := abs(T-O),
        D3 := D0+D1,
        difference(Ts,Os,D3,D).
difference([],[],D0,D) :- true | D = D0.

prep0(N,Seq,LastI,Input,LastT,Target,Dif,Output) :- Dif >= 0.1 |
        Input = [LastI|Input1],
        Target = [LastT|Target1],
        prepareIO(N,Seq,LastI,Input1,LastT,Target1,Output).
prep0(N,Seq,_,Input,_,Target,Dif,Output) :- Dif < 0.1 |
        getOne(Seq,Seq1,Input1,Target1),
        prepIO1(N,Seq1,Input1,Target1,Input,Target,Output).

prepIO1(N,Seq,Input0,Target0,Input,Target,Output) :- Input0 =\= end |
        Input = [Input0|Is],
        Target = [Target0|Ts],
        prepareIO(N,Seq,Input0,Is,Target0,Ts,Output).
prepIO1(_,Seq,end,end,Input,Target,_) :- true |
                Seq = [], Input = [], Target = [].

getOne(Seq0,Seq,Input,Target) :- true |
        Seq0 = [nl,
                write('Input pattern: '), read(Input)| Seq1],
        getOne1(Seq1,Seq,Input,Target).
```

```
getOne1(Seq0,Seq,Input,Target) :- Input ¥= end |
        Seq0 = [
                write('Target pattern: '), read(Target)| Seq].
getOne1(Seq0,Seq,end,Target) :- true | Seq0 = Seq, Target = end.

model(N1,N2,In,Out,Target) :- true |
        distribute(In,Out0),
        layer1(N1,Out0,In1),
        level1(In1,Out1),
        layer2(N2,Out1,In2),
        level2(In2,Out,Target1),
        distribute(Target,Target1).

level1([I|Is],Out) :- true |
        Out = [(O,FB)|Os],
        unit1(I,O,FB),
        level1(Is,Os).
level1([],Out) :- true | Out = [].

unit1(In,Out,FB) :- In = [[_|_]|_] |
        sum1(In,0,Net,In1,Delta),
        O := 1/(1+exp(-Net)),
        Out = [O|Os],
        FB = [FB1|FBs],
        Delta := FB1 * Net*(1-Net),
        unit1(In1,Os,FBs).
unit1([[]|_],Out,_) :- true | Out = [].

level2([I|Is],Out,[T|Ts]) :- true |
        Out = [O|Os],
        unit2(I,O,T),
        level2(Is,Os,Ts).
level2([],Out,[]) :- true | Out = [].

unit2(In,Out,[T|Ts]) :- In = [[_|_]|_] |
        sum1(In,0,Net,In1,Delta),
        O := 1/(1+exp(-Net)),
        Out = [O|Os],
        Delta := (T-O) * Net*(1-Net),
        unit2(In1,Os,Ts).
unit2([[]|_],Out,[]) :- true | Out = [].
```

```
sum1([[(First,Feedback)|Rest]|Is],Sum0,Sum,R,Delta) :- true |
        Feedback = Delta,
        Sum1 := Sum0+First,
        R = [Rest|Rs],
        sum1(Is,Sum1,Sum,Rs,Delta).
sum1([],Sum0,Sum,R,_) :- true | Sum = Sum0, R = [].

layer1(N,[I|Is],Out) :- true |
        makeArc(I,N,I,Out,Out1,_),
        layer1(N,Is,Out1).
layer1(_,[],Out) :- true | setNilsAll(Out).

layer2(N,[(I,FB)|Is],Out) :- true |
        makeArc(I,N,I,Out,Out1,Feedbacks),
        total(Feedbacks,FB),
        layer2(N,Is,Out1).
layer2(_,[],Out) :- true | setNilsAll(Out).

setNilsAll([X|Y]) :- true |
        X = [],
        setNilsAll(Y).
setNilsAll([]) :- true | true.

total(Nums,Sum) :- Nums = [[_|_]|_] |
        Sum = [S|Ss],
        total1(Nums,0,S,Nums1),
        total(Nums1,Ss).
total([[]|_],Sum) :- true | Sum = [].

total1([[N|R]|Ns],Sum0,Sum,Rest) :- true |
        Rest = [R|Rs],
        Sum1 := Sum0+N,
        total1(Ns,Sum1,Sum,Rs).
total1([],Sum0,Sum,Rest) :- true |
        Sum = Sum0,
        Rest = [].
```

```
makeArc(W,N,In,Out,Out1,Feedback) :- N > 0 |
        Out = [[O1|O]|Os],
        Out1 = [O|Os1],
        Feedback = [FB|FBs],
        arc(W,In,O1,FB),
        N1 := N - 1,
        W1 := W * 1.1,
        makeArc(W1,N1,In,Os,Os1,FBs).
makeArc(_,0,_,Out,Out1,Feedback) :- true | Out = [], Out1 = [], Feedback = [].

arc(W,[I|Is],Out,Feedback) :- true |
        Out = [(O,D)|Os],
        O := W * I,
        updateWeight(W,I,D,W1),
        Feedback = [FB|FBs],
        FB := W * D,
        arc(W1,Is,Os,FBs).
arc(_,[],Out,FB) :- true | Out = [], FB = [].

updateWeight(Old,Input,Delta,New) :- wait(Delta) |
        K = 0.5,
        New := Old + K * Delta * Input.

distribute([First|Rest],Out) :- true |
        dist1(First,Out,Out1),
        distribute(Rest,Out1).
distribute([],Out) :- true | setNilsAll(Out).

dist1([I|Is],O,R) :- true |
        O = [[I|Rest]|Os],
        R = [Rest|Rs],
        dist1(Is,Os,Rs).
dist1([],O,R) :- true | O = [], R = [].

constructOutput(I,O) :- I = [[_|_]|_] |
        gatherFirstAll(I,O1,I1),
        O = [O1|Os],
        constructOutput(I1,Os).
constructOutput([[]|_],O) :- true | O = [].
```

```
gatherFirstAll([[First|Rest]|Others],Pack,Continue) :- true |
        Pack = [First|Fs],
        Continue = [Rest|Rs],
        gatherFirstAll(Others,Fs,Rs).
gatherFirstAll([],Pack,Continue) :- true |  Pack = [], Continue = [].
```

付録２　シンタクスシュガーによる記述例

```
net(In,Out,Target) :-
        layer1(In,M1),
        hiddenLevel(M1,M2),
        layer2(M2,M3,Error),
        outputUnit(M3,Out,Target,Error).

layer1((In1,In2),(W1,W2,W3,W4,W5,W6),Out) :- true |
        link(In1,W1,Out1),
        link(In1,W2,Out2),
        link(In1,W3,Out3),
        link(In2,W4,Out4),
        link(In2,W5,Out5),
        link(In2,W6,Out6),
        Out=((Out1,Out4),(Out2,Out5),(Out3,Out6)).

layer2((In1,In2,In3),Out,Error) :- true |
        link(In1,W1,Out1,Error),
        link(In1,W2,Out2,Error),
        link(In1,W3,Out3,Error),
        Out=(Out1,Out2,Out3).

outputUnit(In,Out,Target,Error) :- true |
        add3(In,Net),
        makeOutput(Net,Out),
        getOutputError(Net,Out,Target,Error).

makeOutput((In1?,In2?,In3?),Out^) :-
        Out is 1/(1+exp(-(In1+In2+In3))).

getOutputError(Net?,Out?,Target?,Error^) :-
        Error is (Target-Out)*Net*(1-Net).

link(In,W,Out,Error) :- true |
        weighten(In,[W|Ws],Out),
        getNewWeight(W,Error,In,Ws).

getNewWeight(Old,[E1|Es],[In1|Ins],W) :- true |
        W1 := W+0.5*E1*In1,
        W=[W1|Ws],
        getNewWeight(W,Es,Ins,Ws).
```

```
link(X?,W?,Out) :-
        Out is X*W.

hiddenLevel((In1,In2,In3),Out) :- true |
        hiddenUnit(In1,Out1),
        hiddenUnit(In1,Out2),
        hiddenUnit(In3,Out3),
        Out=(Out1,Out2,Out3).

hiddenUnit((In1?,In2?),Out^) :-
        Out is 1/(1+exp(-(In1+In2))).

add3((In1?,In2?,In3?),Out^) :-
        Out is In1+In2+In3.
add3((In1?,In2?),Out^) :-
        Out is In1+In2.
```