`

**SHERLOCK — Neural Network Software for Automated Problem Solving**

**Qiming Zhang**
**Final Year Project**
**BSc (Hons) Applied Computing**
**University of Dundee, 2011**
**Supervisor: Dr Katya Komendantskaya**

***Abstract* –** This report aims to apply methods of neural-symbolic integration which integrates connectionist systems and symbolic AI systems; and capitalize on the advantages of the two paradigms. There are two types of neural-symbolic systems – Tp neural networks and CILP systems both of which are capable to do deduction. We want to use neural-symbolic systems to solve logic deduction problem. We develop an interface called SHERLOCK that implements these systems and solves logic deduction problems. We test our implementation on several interesting problems.

## 1 Introduction

The detective Sherlock Holmes is famous for his astute logical reasoning and his observational skills to solve difficult cases. One kind of logical reasoning is deductive reasoning, which constructs or evaluates deductive conclusions from facts and knowledge.

Here is a crime detective story: A small bank was robbed. The bank safe is found open. Footprints left on the floor belong to a person with a pair of small feet; and burned cigarettes are found on the scene. A detective came up with a deduction rule-"If someone has keys for bank safe, small feet and the habit of smoking, the one is the criminal." Then the detective found the following facts:

1) "Jane, Harry and Stephen have keys."
2) "Stephen and Jane have small feet."
3) "Stephen has a habit of smoking."

The detective drew the conclusion logically that Stephen was the criminal.

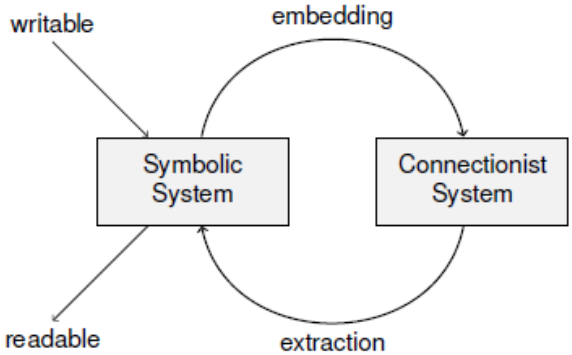
The story above is a simple example to show what deductive reasoning is. In deductive logic, a conclusion necessarily follows from a set of premises. This article will introduce two types of neural networks and the deductive ability of the neural networks to solve logic problems.

In practice, deduction may be more complicated and uncertain. Consider a police officer who has just come to a crime scene and wishes to make use of all evidence available to find suspects and rule out the suspension of unrelated people. To be efficient, the police officer uses a small portable computer that has a problem-solving assistant. What should this assistant be like? A neural network software would come in handy, because it has been trained with sufficient examples so that it is capable to do deduction with incomplete information; also - it is very efficient to assist the police officer do his job. This could be achieved by deploying a neural-symbolic system.

[1]"Human cognition successfully integrates the connectionist and symbolic paradigms of Artificial Intelligence (AI). Yet modelling of cognition develops these separately in neural computation and symbolic logic/AI areas. There is now a movement towards a fruitful mid-point between these extremes, in which the study of logic is combined with connectionism. ".

Connectionist systems and symbolic AI systems have quite contrasting advantages and disadvantages. It is very significant to put forward Neural-symbolic Integration which integrates both paradigms while retaining all the advantages. There are sound, well-developed methods and techniques to build such systems.

To design Neural-Symbolic Learning systems, it is necessary to use logic programming to present symbolic knowledge. Then, one needs translation methods to interpret symbolic knowledge to connectionist models, and to make sure that the network obtained is equivalent to the original program, in the sense that what is computed by the program is computed by the network. With such neural networks, we can do deduction. The following figure shows the neural symbolic cycle.




With respect to translation methods, two models will be introduced in the following, either of which provides a correct translation algorithm. One is a single-hidden-layer partially recurrent neural network with binary threshold units, having the ability to compute the least fixpoint of a general logic program, introduced in *Massively Parallel Model for Logic Programming, see* [2]. Because these

`

neural networks correspond to fixpoint operator Tp for logic programs, we will call them Tp-neural networks. The other is *Connectionist Inductive Learning and Logic Programming* (CILP), a neural-symbolic learning system based on a feedforward neural network that integrates inductive and deductive learning, see [1].

For a real logic deduction problem, we will use logic programming to formalise deductive inference of the logic problem e.g. a crime-solving problem. Once the deductive reasoning is expressed as a logic program *P*, there are two translation algorithms to transform *P* into a Tp- or CILP-neural network N. There are theorems [2, 1], guaranteeing that the neural network *N* can compute the fixpoint operator Tp of *P*. The fixed point can be interpreted as the answer to the deduction problem.

**Objectives**
In the main body of this report, we address the four objectives. Firstly, we design a logic programming interface for users who use neural-symbolic systems to operate deduction tasks. Secondly, we show one way to build up a knowledge refining system. Thirdly, we solve the world-famous logic deduction problem – Einstein's Riddle with the use of our logic programming interface. Finally, we carry on an experiment, designing a new type of neural networks, which is a derivative from Tp neural networks and enables classical negation in logic programming.

**Methods**
The first objective, to design a logic programming interface for neural-symbolic systems, is the most important work in this report. Therefore it is worthy and necessary to give an introduction to it here.

The neural-symbolic systems are capable to do deduction. However, it is very difficult for users to use such a system because users have to insert symbolic knowledge into a neural network by setting up weights and thresholds and then interpret the outcome of the neural network to symbolic knowledge according to users' definition to neurons. An interface between users and neural-symbolic systems is required.

The interface is supposed to provide a logic programming code editor which is writable in terms of embedding symbolic knowledge into connectionist systems and a deduction outcome reader which is readable in terms of interpreting the outcome of neural-symbolic systems to symbolic knowledge.

The detailed specification is shown as follows:

1. This interface should provide a declarative language for logic programming, which would better to be similar to the well-known logic programming – Prolog.
2. This interface should be able to build up neural-symbolic systems (Tp- and CILP-neural networks) and embed the symbolic knowledge according to a general logic program.
3. This interface should ensure the neural-symbolic systems can compute the fixed point of a well-behaved logic program.
4. This interface should be able to interpret the outcome of a neural-symbolic system to symbolic knowledge and present the knowledge in a form of the declarative language.

**Software**
We deployed Microsoft Visual C# for developing SHERLOCK. Matlab is used for additional tests.

The report is organised as follows. Section 2 contains background knowledge. Section 3 describes the design and implementation of a logic programming interface - SHERLOCK. Section 4, we show how to design a rule-based neural network for classification and probability estimation. Section 5 is the test – Einstein's Riddle on SHERLOCK. In Section 6, we introduce the experiment on classical negation.

# 2 Background

In the following subsections, we briefly recall basic notions and notations concerning logic programs, Tp- and CILP- neural networks, but we restrict ourselves to logic deduction. In each subsection, we would briefly show how the knowledge will be used.

## 2.1 Definite Logic Programs: the syntax

In this section, we give a concise introduction to Definite Logic Program because it is very fundamental and important.

**Definition 2.1.1** In definite logic, there are five classes of symbols called *alphabet*:

a) Variables symbols, x, y and z, …
b) Constants symbols, a, b and c, …
c) Predicate symbols, p, q and r, …
d) Function symbols, f, g and h, …
e) Connectives, which are ∧ ("and"), ← ("if-then"), ~ ("not").
f) Punctuation symbols, which are "(", ")" and ",".

**Definition 2.1.2** A *term* is defined inductively as follows:

(a) A variable is term.
(b) A constant is term.
(c) If f is an a-ary function symbol and $t_1,\ldots,t_n$ are terms, then $f(t_1,\ldots,t_n)$ is a term.

**Definition 2.1.3** An *atomic formula* (or an *atom*) is defined inductively as follows: If p is an n-ary predicate symbol and $t_1,\ldots,t_n$ are terms, then $p(t_1,\ldots,t_n)$ is a atom.

`

**Definition 2.1.4** If A is an atomic formula then

- A is said to be a **positive *literal***
- ~A is said to be a **negative *literal***

**Definition 2.1.5** A *definite program clause* is a clause of the form

$$\underbrace{A}_{\text{head}} \quad \leftarrow \quad \underbrace{L1, \ \dots \ , \ Ln}_{\text{body with Li either X or } \sim X}$$

, which contains precisely one atom (viz. A) in its consequent. A is called the *head* of the clause, the Li are called *body literals*, and their conjunction $L_1, \dots, L_n$ is called the *body* of the clause.

**Definition 2.1.6** A definite program is a finite set of definite program clauses.

A logic program is a very necessary tool to present a practical problem in a logic way.

**Example 2.1.1** *Consider the story in Introduction, there is a logic program which consists of clauses (1) to (7) as follows:*

*(1) Criminal(X)←*
*HasKeys(X), SmallFeet(X), Smoke(X)*
*(2) HasKeys(Harry)←*
*(3) HasKeys(Jane)←*
*(4) HasKeys(Stephen) ←*
*(5) SmallFeet(Stephen) ←*
*(6) SmallFeet(Jane)←*
*(7) Smoke(Stephen)←*

Clause (1) is a rule, representing "if some has keys for bank safe, small feet and the habit of smoking, the one is the criminal."
Clauses (2)-(7) are the facts, interpreted as follows: 1) "Jane, Harry and Stephen have keys." 2) "Stephen and Jane have small feet." 3) "Stephen has a habit of smoking."

This introduction to Definite Logic Program prepares the fundamental knowledge for introducing the following knowledge.

## 2.2 Semantics of Definite Logic Programs

In this section, we give Semantics to Definite Logic Program and Fixed-point Theory.

**Definition 2.2.1** Let P be a definite program. The *Herbrand universe*, denoted $U_p$, is the set of all ground terms that can be formed from the constants and function symbols appearing in.

**Definition 2.2.2** Let P be a definite program. The *Herbrand base*, denoted $B_p$, is the set of all ground atoms that can be formed from the predicates in *P* and the terms in the Herbrand universe.

**Definition 2.2.3** Let P be a definite program. A *Herbrand interpretation* denoted I, is a subset of the Herbrand base.

An interpretation can map from atoms in $B_p$ to {true, false}.An interpretation is extended to map clauses in P to {true, false} as follows: a definite clause A← $L_1$, ..., $L_n$ is mapped to true either A is mapped to true or if one of Li is mapped to false.

**Definition 2.2.4** Let P be a definite program. A Herbrand mode is a Herbrand interpretation that maps all clauses in P to true.

**Definition 2.2.5** Let P be a definite program. The least Herbrand model of P, denoted $M_p$, is the smallest subset of $B_p$ that represents an interpretation that is a model of P.

**Example 2.1.1** Considering the logic program in Example 2.1.1:

- The Herbrand Universe is $U_L$ = {Jane, Harry and Stephen}.
- The Herbrand Base is $B_p$ = { *Criminal(*Jane*), Criminal(*Harry*), Criminal(*Stephen*), HasKeys(*Jane*), HasKeys(*Harry*), HasKeys(*Stephen*), SmallFeet(*Jane*), SmallFeet(*Harry*), SmallFeet(*Stephen*), Smoke(*Jane*), Smoke(*Harry*), Smoke(*Stephen*)*}.
- $I_1$ = {*HasKeys(Harry), HasKeys(Jane), HasKeys(Stephen), SmallFeet(Stephen), SmallFeet(Jane),Smoke(Stephen), Criminal(Stephen)* } is a Herbrand interpretation.
- $I_2$ = { *HasKeys(Jane) ,SmallFeet(Jane)*} is a Herbrand interpretation.
- $I_1$ is a Herbrand model for this logic program while $I_2$ is not. $I_2$ maps the clause "*Criminal(Jane) ← HasKeys(Jane), SmallFeet(Jane), Smoke(Jane)" to false.*
- $I_1$ is also the *least Herbrand* model for this logic program.

Let P be a definite program. The collection of Herbrand interpretations forms a complete lattice and there is a monotonic mapping associated with P defined on this lattice. Some concepts and results about a complete lattice are showing as follows.

**Definition 2.2.6** Let *[L,⩽]* be a complete lattice and *T: L → L* be a mapping. *a ∈ L* is the least fixpoint of *T if a* is fixpoint of *T (i.e. T(a)=a )* and for all fixpoints *b of T, a ⩽ b.* Similarly, *a ∈ L* is the greatest fixpoint of *T* if *a* is *a* fixpoint of *T* and for all fixpoints *b of T, b ⩽ a.*

**Proposition 2.2.1** Let *[L,⩽]* be a complete lattice and *T: L → L* be monotonic. T has a least fixed point *(lft(T))* and a greatest fixed point *(gfp(T)).*

**Definition 2.2.7** Let $L$ be complete lattice and $T: L \rightarrow L$ be monotonic. Then we define:

- $T \uparrow 0 = inf(L)$; *inf(L) is the greatest lower bound of* $L$.
- $T \uparrow \alpha = T(T \uparrow (\alpha - 1))$; *if* $\alpha$ *is an ordinal.*

Let $P$ be a definite program. Then $2^{Bp}$, which is the set of all Herbrand interpretations of $P$, is a complete lattice under the partial order of set inclusion $\subseteq$.The top element of this lattice is $B_p$ and the bottom element is $\emptyset$.

**Definition 2.2.8** Let P be a definite program. The mapping $Tp: 2^{Bp} \rightarrow 2^{Bp}$ is defined as follows. Let I be a set Herbrand Interpretation; then $Tp\ (I) = \{A \in B_p | A \leftarrow A_1, \ldots, A_n$ *is a clause in P* and $\{ A_1, \ldots, A_n \} \subset I\}$.

The Tp-operator propagates truth along the clauses. In other words, *Tp* provides the link between the declarative and the procedural semantics of *P*. For definite programs, TP converges to the least model. Therefore, Herbrand interpretations that are models can be characterised in terms of *Tp*.

**Proposition 2.2.1** Let $P$ be a definite program and I a Herbrand interpretation of $P$. Then the mapping Tp is continuous and I is a model of $P$ *iff* $Tp\ (I) \subseteq I$.

**Proposition 2.2.2** Let $P$ be a definite program. $M_p = lfp(Tp) = Tp \uparrow w$.

**Example 2.2.2** Considering the logic program in Example 2.1.1:

1) $Tp \uparrow 0 = \emptyset$ ;
2) $Tp \uparrow 1$={*HasKeys(Harry), HasKeys(Jane), HasKeys(Stephen), SmallFeet(Stephen), SmallFeet(Jane),Smoke(Stephen)*}
3) $Tp \uparrow 2$ = {*HasKeys(Harry), HasKeys(Jane), HasKeys(Stephen), SmallFeet(Stephen), SmallFeet(Jane),Smoke(Stephen), Criminal(Stephen)* }
4) $Tp \uparrow 2 = Tp \uparrow 3 = Tp \uparrow w$

**Note** As Bp is finite, the lattice is also finite, and there is some n ϵ w such that Tp ↑ n = Tp ↑ n+1, and hence Tp ↑ w. will be equal to Tp ↑ n, for some successor ordinal n ϵ w.

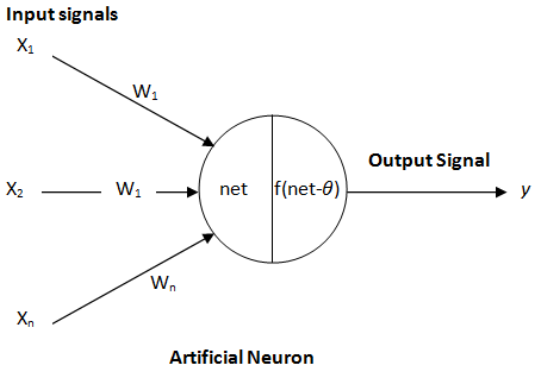
Semantics of Definite Logic program give a means to present the knowledge of a logic program P in a purely syntactic way. Fixed-point Theory gives a means to compute the least Herbrand model of P according to the very initial knowledge of P.

### 2.3 Neural Networks

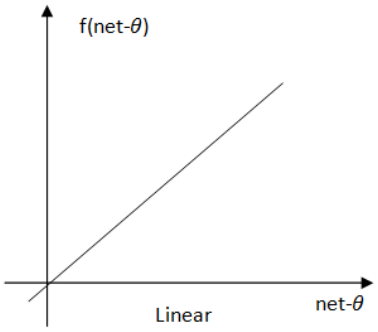
Here we make a brief induction to Artificial Neural Networks (ANNs) because it is also very fundamental and important.

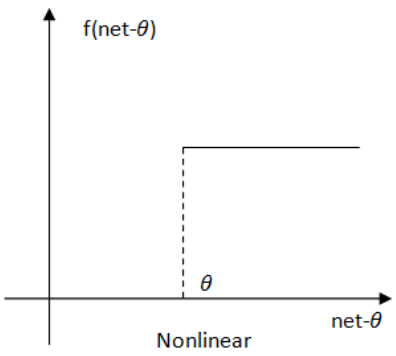
An artificial neural network is a massively parallel computational model. It is inspired from neuroscience research that tries to imitate some key behaviour of animal brains (biological neural networks).

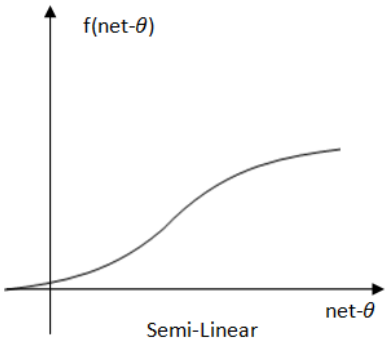
**Definition 2.3.1** An *artificial Neuron* (AN) is a model of a biological neural (BN). Each AN receives a vector of $I$ input signals $\vec{X} = (x_1, x_2, \ldots, x_n)$ from the environment or other ANs. To each input signal $x_i$ is associated a weight $W_i$ to strengthen or deplete the input signals. The An computes the net input signal *net* , and uses an activation function $f$ to compute the output signal $y$ given the net input *net* and a threshold value $\theta$, also referred to as the bias.




The net input signal to an AN is usually computed as the weighted sum of all input signals, net $= \sum_{i=1}^{I} x_i w_i$. The activation function f receives the net input and the bias, and determines the output of the neuron. There are three basic activation functions: linear, non-linear and semi-linear.

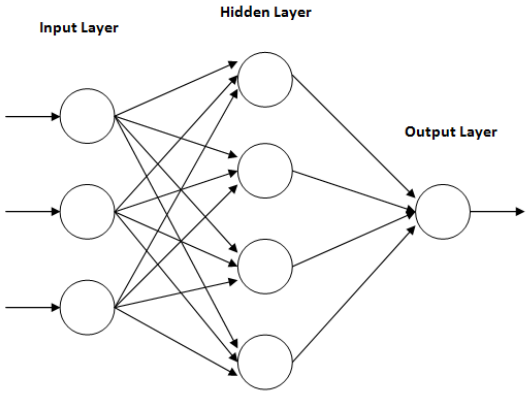





**Definition 2.3.2** An *artificial neural network* (ANN) is a layered network of ANs. A typical *artificial neural network* consists of an *input layer*, *hidden layers* and an *output layer*. ANs in one layer are connected, fully or partially, to the ANs in the next layer. Feedback connections to ANs in previous layers are also possible.

ANs in input layer don't have input connections and ANs in output layer don't have output connections. Only hidden neurons have both input and output connection.




A typical 3-layer Artificial Neural Network

**Definition 2.3.3** A feedforward neural network is an artificial neural network where connections between the neurons do not form a directed cycle. This is different from recurrent neural networks. The network above is a feedforward neural network.

**Definition 2.3.4** Feedforward operation is a computational procedure that a feedforward neural network takes in an input vector and produces an output vector.

In [9], for a d-$n_H$-c fully connected three-layer network, it has d neurons in the input layer, $n_H$ neurons in the hidden layer and c neurons in the output layer. During feedforward operation, a d-dimensional input vector $\vec{x}$ is presented to the input layer; each input neuron then emits its correspondent xi. Each of the $n_H$ hidden neurons computes its net, $net_j$ as the inner product of the input layer signas with weight $w_{ji}$. The hidden neuron emits $y_j=f(net_j - \theta_j)$, where f is the activation and $\theta_j$ is the threshold. Each of the c output neurons computes it net, $net_k$ as the inner the product of the hidden layer signals with weight $w_{kj}$. The output neuron emits $z_k=f(net_k-\theta_k)$, where f is the activation function in the output layer and $\theta_k$is the threshold. The feedforward operation is defined inductively as follows:

1. For $j\in[1,\ldots,n_H]$, $\text{net}_j = \sum_{i=1}^{d} X_i W_{ji}$
2. For $j\in[1,\ldots,n_H]$, $y_j = f(net_j - \theta_j)$
3. For $k\in[1,\ldots,c]$, $net_k = \sum_{j=1}^{n_H} y_j w_{kj}$
4. For $k\in[1,\ldots,c]$, $z_k = f(net_k - \theta_k)$

Neural networks are introduced here because the following two types of neural-symbolic systems need a connectionist system to embed symbolic knowledge.

### 2.4 Tp-Neural Network

We now introduce Tp–neural network that can simulate the Tp-operator. It is titled as the Core-method for Propositional Logic in neural-symbolic integration.

**Theorem 2.4.1** [1]For each program P there exists a single hidden layer feedforward network N of binary Threshold units such that *N* computes Tp. ( Holldobler & Kalinke, 1994)

**Theorem 2.4.2 [1]**For each single hidden layer feedforward network N of binary Threshold units there exists a program P such that Tp is computed by N. ( Holldobler & Kalinke, 1994)

Holldobler and Kalinke also presented a translation algorithm to convert a general definite logic problem to a single hidden layer neural network. This kind of network is called a Tp-neural network.

**[2]Translation Algorithm**

Let p and q be the number of ground atoms and the number of clauses occurring in P, respectively. Without loss of generality we may assume that the grounded atoms are numbered from 1 to p. The network associated with P can now be constructed as follows:

1) The input and output layer is a vector of p binary threshold neurons, where the *i*th neuron represents the atom $A_i$, $1\leq \mathbf{i} \leq p$. The threshold of each neuron occurring in the input or output layer is set to 0.5.
2) For each clause of the form $A\leftarrow A_1 ,\cdots, \mathbf{A}_m, \sim A_{m+1} ,\cdots, \sim An(0\leq \mathbf{m} \leq n)$,occurring in P do the following:
   a) Add a binary threshold unit c to the hidden layer of N.
   b) Connect c to the unit representing A in the output layer with weight 1.
   c) For each $A_i$, $1\leq \mathbf{i} \leq m$, connect the unit representing $A_i$ in the input layer of N to c, and set the connection's weight to 1.

`

d) For each ~$A_i$, m+1≤ **i** ≤n, connect the unit representing $A_i$ in the input layer of N to c, and set the connection's weight to -1.
e) Set the threshold $\boldsymbol{\theta_c}$ of c to m-0.5.

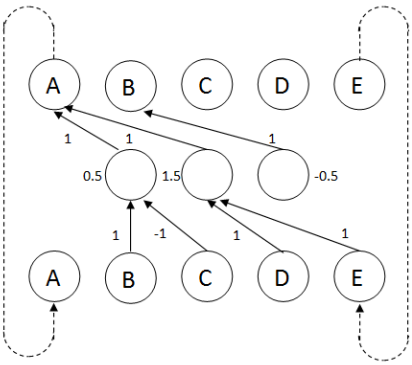
Note: For all artificial neurons, their activation function is a non-linear function:

$$f(x) = \begin{cases} \mathbf{1} & , \boldsymbol{if\ x \geq 0.5} \\ \mathbf{0} & , \boldsymbol{ohters} \end{cases}$$

A theorem then shows that the network obtained is equivalent to the original logic program, in the sense that what is computed by the program is computed by the network and vice versa. That means the network can compute out the least Herbrand model of the original logic program.

**Example 2.4.1** Consider the following logic program P:

1) A ←B, ~C
2) A ← D,E
3) B ←




The above neural network N is obtained by applying the above algorithm to P. If B = 1 and C =0 in the input layer of N then A = 1 in the output layer of N, representing the rule A ← B ~ C of P. Similarly, B = 1 is always obtained in the output layer of N, regardless of its input vector, indicating that B is a fact.

According to the above two theorems, each interpretation I for P is represented.

Let us recall that essentially, Tp-operator is a mapping from the set of all *Herbrand interpretations of P* to the set of all *Herbrand interpretations of P*, that is, Tp*:* $2^{Bp} \rightarrow 2^{Bp}$. Now the network itself is a mapping from the set of all *interpretations of Herbrand base* to the set of all *interpretations of Herbrand base.*

If we iterate the network by putting output neurons interpretations to input neurons interpretations in the same way that the Tp-operator iterate itself with *Herbrand Interpretation*s, the network then will converge to a static state that the output neurons interpretations are the least Herbrand Model.

**Theorem 2.4.3** [1]Let P be a well-behaved program. There exists a single hidden layer recurrent network such that each computation starting with an arbitrary initial input I converges to a stable state and yields the unique fixpoint of Tp.

Here is the algorithm to compute the least Herbrand Model of P in a Tp-neural network.

**Massively Parallel Deduction Algorithm**

Let p be the number of input neurons and the number of output neurons in Tp-neural network N. The input is defined by a vector I = ($I_1$, … , $I_p$) and the output is given by a vector O = ($O_1$, . . . ,$O_p$).
1. Initialize I = [0, 0, …, 0].
2. Loop:
   1) Calculate O = feedforward(I);
   2) If I is equal to O, then terminate;
   3) If I is not equal to O, then for j in (1, …, p), replace the value of $I_j$ with the value of $O_j$ in O.

**Example 2.4.2** Consider the logic program in Example 2.4.1.As shown in Figure, N is recurrently connected, its output vector feeds the input vector in an iteration of Tp. Let the initial input vector I = (0, 0, 0, 0, 0). So the deduction process would be as follows:

*(1)* $Tp \uparrow 0 = \{0, 0, 0, 0, 0\}$;
*(2)* $Tp \uparrow 1 = \{0, 1, 0, 0, 0\}$;
(3) $Tp \uparrow 2 = \{1, 1, 0, 0, 0\}$;
*(4)* $Tp \uparrow 2 = Tp \uparrow 3 = Tp \uparrow w$ .

**Note** [3]There are four characteristic properties that distinguish TP -neural networks. We summarise them as follows.
1. The number of neurons in the input and output layers is the number of atoms in the Herbrand base BP of a given program P.
2. The number of iterations of TP (for a given logic program P) corresponds to the number of iterations of the neural network built upon P.
3. Signals of TP -neural networks are binary, and this is achieved by using binary threshold activation functions. This provides the computations of truth value functions and that are used in program clauses.
4. As a consequence of the property 3, first-order atoms are not presented in the neural network directly, and only truth values 1 and 0, which are the same for all the atoms, are propagated.

Tp-neural networks are introduced to transform a logic program to a neural network which is proved to have the ability to approximate the fixpoint operator of the corresponding program and compute out the fixed point.

`

## 2.5 CILP-Neural Networks

In this section, we introduce a massively parallel computational model that can simulate the Tp-operator. It is a derivative from the Core-Method for Propositional Logic in neural-symbolic integration.

[5]*Connectionist Inductive Learning and Logic Programming* (CILP) is a massively parallel computational model based on a feedforward artificial neural network that integrates inductive learning from examples and background knowledge with deductive reasoning using logic programming.

To embed the propositional knowledge of a general logic program (P) in a neural network (N), CILP uses an approach similar to Tp-Neural Networks. However, A CILP-neural network N deploys a semi-linear function as its activation function:

$$f(x) = 2/(1+e^{-\beta x})-1.$$

Clearly f(x) has the real numbers as domain and [-1, 1] as codomain.

**Theorem 2.5.1** [1] For each propositional general program P, there exists a feedforward artificial neural network N with exactly one hidden layer and semi-linear neurons such that N computes Tp.

The translation algorithm that converts a general logic program to a CILP-neural network is shown as follows.

**[1]Translation Algorithm**

Notation Given a general logic program *P*, let:

1. $q$ denote the number of clauses $C_l$ $(1 \le l \le q)$ occurring in *P*;
2. $A_{min}$, the minimum activation for a neuron to be considered *active* (or *true*), $0 < A_{min} < 1$;
3. $A_{max}$, the maximum activation when a neuron is not active (or *false*), $-1 < Amax < 0$ ;
4. $h(x) = \frac{2}{1+e^{-\beta x}} - 1$, the bipolar semi-linear activation function;
5. $g(x) = x$, the standard linear activation function;
6. $W$ (resp. $-W$), the weights of connections associated with positive (resp. negative) literals;
7. $\theta_l$ denote the threshold of the hidden neuron $N_l$ associated with clause $C_l$ ;
8. $\theta_A$ denote the threshold of the output neuron $A$, where $A$ is the head of clause $C_l$ ;
9. $k_l$ denote the number of literals in the body of clause $C_l$ ;
10. $\mu_l$ denote the number of clauses in P with the same atom in the head, for each clause $C_l$ ;
11. $MAX_{Cl}$ $(k_l, \mu_l)$ denote the greater element of $k_l$ and $\mu_l$ for clause $C_l$ ;
12. $MAX_P(k_1, \ldots, k_q, \mu_1, \ldots, \mu_q)$ denote the greatest element of all ks and μs of P.

Main body of Translation Algorithm:

1. Given a general logic program P, there are n propositional variables. Then there are n input neurons and n output neurons in a neural network N. Label each input (resp. output) neuron of the neural network with each of these propositional variables. Assume that $A_{max} = - A_{min}$.
2. Calculate $MAX_P(\vec{k}, \vec{u})$ of P;
3. Calculate $A_{min} > \frac{MAX_p(\vec{k},\vec{u})-1}{MAX_p(\vec{k},\vec{u})+1}$ ;
4. Calculate the value of $W$ such that the following is satisfied:

$$W \ge \frac{2}{\beta} * \frac{ln\,(1+Amin)-ln\,(1-Amin)}{MAX_p(\vec{k},\vec{u})*(Amin-1)+Amin+1}$$

5. For each clause $C_l$ of $P$ of the form $A \leftarrow L_1, \ldots, L_k$ $(k \ge 0)$:
   a. Create input neurons $L_1, \ldots, L_k$ and an output neuron $A$ in $N$ (if they do not exist yet).
   b. Add a neuron $N_l$ to the hidden layer of $N$.
   c. Connect each neuron $L_i$ $(1 \le i \le k)$ in the input layer to the neuron $N_l$ in the hidden layer. If $L_i$ is a positive literal, then set the connection weight to $W$; otherwise, set the connection weight to $-W$.
   d. Connect the neuron $N_l$ in the hidden layer to the neuron $A$ in the output layer and set the connection weight to $W$.
   e. Define the threshold $(\theta_l)$ of the neuron $N_l$ in the hidden layer as

$$\theta_l = \frac{(1+A_{min})(k_l-1)}{2} W.$$

   f. Define the threshold $(\theta_A)$ of the neuron $A$ in the output layer as

$$\theta_A = \frac{(1+A_{min})(1-\mu_l)}{2} W.$$

6. For those neurons in the output layer that don't have a threshold yet, define the threshold $(\theta_o)$ of each of those neurons in the output layer as $\theta_o = \frac{(1+A_{min})}{2} W$ .
7. Set g(x) as the activation function of the neurons in the input layer of N. In this way, the activation of the neurons in the input layer of N given by each input vector i will represent an interpretation for P.
8. Set $h(x)$ as the activation function of the neurons in the hidden and output layers of $N$. In this way, a gradient descent learning algorithm, such as backpropagation, can be applied to $N$.
9. If $N$ needs to be fully connected, set all other connections to zero.

**Example 2.5.1** Consider the logic program in Example 2.4.1. The CILP network should be set up as follows:

$MAX_P(\vec{k}, \vec{u}) = 2$.

$A_{min} > 1/3$; let $A_{min} = 0.5$.
$W \geq 4.394/\beta$. Let $\beta = 1$ and $W = 4.5$.
*For clause 1, A ←B, ~C:*

$$\theta_1 = \frac{(1+A_{min})}{2} W = 3.375;$$

$$\theta_A = -\frac{(1+A_{min})}{2} W = -3.375.$$

*For clause 2, A ←D, E:*

$$\theta_2 = \frac{(1+A_{min})}{2} W = 3.375;$$

$$\theta_A = -\frac{(1+A_{min})}{2} W = -3.375.$$

*For clause 3, A ←D, E:*

$$\theta_3 = -\frac{(1+A_{min})}{2} W = -3.375;$$

$$\theta_B = 0.$$

*Neuron C has a threshold*

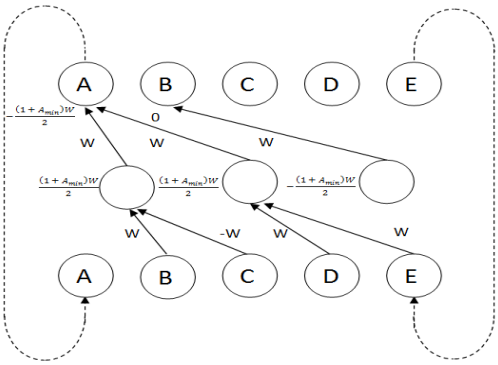
$$\theta_C = \frac{(1+A_{min})}{2} W = 3.375.$$

*Neuron D has a threshold*

$$\theta_D = \frac{(1+A_{min})}{2} W = 3.375.$$

*Neuron E has a threshold*

$$\theta_E = \frac{(1+A_{min})}{2} W = 3.375.$$




If B = 1 and C =-1 in the input layer of N then A = 0.9992 in the output layer of N, representing the rule A ← B ~ C of P. Similarly, B = 0.9705 is always obtained in the output layer of N, regardless of its input vector, indicating that B is a fact.

In order to perform deduction, a CILP –neural network N is transformed into a partially recurrent network N* by connecting each neuron in the output layer to its correspondent neuron in the input layer with weight 1, as shown in figure. In this way, N* can iterate Tp in parallel.

[1]**Massively Parallel Deduction Algorithm**

Let p be the number of input neurons and the number of output neurons in Tp-neural network N. The input is defined by a vector $I = (I_1, \ldots, I_p)$ and the output is given by a vector $O = (O_1, \ldots, O_p)$.

1. Initialize I = [-1, -1, …, -1].
2. Loop:
   1) Calculate O = feedforward(I);
   2) Define $\boldsymbol{valuation}(x) = \begin{cases} \mathbf{1}, \mathbf{if}\ x > A_{min}, \\ -\mathbf{1}, \mathbf{otherwise} \end{cases}$ if *valuation*($I_j$) is equal to *valuation* ($O_j$) (j∈[1, …, p]), then terminate;
   3) Otherwise, for j∈[1, …, p], replace the value of $I_j$ with the value of $O_j$.

Let the initial input vector I = (-1, -1, -1, -1, -1). So the deduction process would be as follows:

*(1)* $Tp \uparrow 0$ = { (-1, -1, -1, -1, -1};
*(2)* $Tp \uparrow 1$ = { -0.9903, 0.9705, -0.9338, -0.9338, -0.9338};
(3) $Tp \uparrow 2$ = { 0.9305, 0.9705, -0.9338, -0.9338, -0.9338};
*(4)* $Tp \uparrow 2 = Tp \uparrow 3 = Tp \uparrow w$ .

CILP systems are introduced to transform a logic program to a neural network which is proved to have the ability to approximate the fixpoint operator of the corresponding program and compute out the fixed point. However, compared to a Tp-neural network, a CILP system is more powerful in the sense that it has the capability to do machine learning due to the non-linear activation function. The logic program P is viewed as background knowledge. A CILP network can be trained with examples efficiently, then refining the background knowledge.

## 3 A logic programming interface

*Design*

According to the specification for a logic programming interface in the Introduction Section, this interface is supposed to consist of the following components:

1. A code editor, with which users can present a general logic program in a prolog-like declarative language;
2. A translator, which can analyse syntax and semantics of the logic program and set up neural-symbolic systems according to the logic program;
3. A model of Tp neural networks and a model of CILP-neural networks;
4. An interpreter and a result reader

**Models of neural-symbolic systems**

**A model of Tp-neural networks**

Given a general logic program *P*, let there be *q* definite clauses, *n* propositional variables. A Tp-neural network translated from *P* owns n input neurons, q hidden neurons and n output neurons with a n-by-q matrix W1 – weights between the input layer and the hidden layer and an q-by-n matrix W2 – weights between the hidden layer and the output layer. In the output layer there are n thresholds, whose values are set 0.5. In the hidden layer there are q thresholds, whose values are determined by W1.

`

**Model** For a Tp-neural network, a tuple (*n, q, W1, W2*) determines all the properties of the Tp-neural network. *q* is the number of clauses. *n* is the number of all propositional variables. *W1* is a *n-by-q* matrix; and *W2* is an *q-by-n* matrix. They represent the connections between clauses' bodies and their corresponding heads. The q thresholds $\boldsymbol{\theta}_l$ ($l \in [1,\ldots,q]$) in the hidden layer are calculated as follows:

- For $l \in [1,\ldots,q]$, denote the number of positive value in the $l$-th column of W1 as $p_l$. $\boldsymbol{\theta}_l = p_l - 0.5$.

**A model of CILP-neural networks**

Given a general logic program *P*, let there be *q* definite clauses, *n* propositional variables. A CILP-neural network translated from *P* owns n input neurons, q hidden neurons and n output neurons with a q*n matrix *W1'* – weights between the input layer and the hidden layer and an q*n matrix *W2'*– weights between the hidden layer and the output layer. *W1* is derived if *W1'* divide by $\boldsymbol{W}$; and *W2* is derived if *W2'* divide by $\boldsymbol{W}$. *W1* and *W2* determine W1', W2', thresholds in the hidden layer $\boldsymbol{\theta}_l$ ($l \in [1,\ldots,q]$) and thresholds in the output layer $\boldsymbol{\theta}_o$ ($o \in [1,\ldots,n]$) in the neural-symbolic system.

**Model** For a CILP-neural network, a tuple (*n, q , W1, W2*) determines all the properties of the CILP-neural network. *q* is the number of clauses. n is the number of all propositional variables. W1 is a k*r matrix; and W2 is an r*k matrix. They represent the connections between clauses' bodies and their corresponding heads.

$\boldsymbol{W}$, $\boldsymbol{A_{min}}$, thresholds in the hidden layer $\boldsymbol{\theta}_l$ ($l \in [1,\ldots,q]$) and thresholds in the output layer $\boldsymbol{\theta}_o$ ($o \in [1,\ldots,n]$), $\vec{k}$, $\vec{u}$ are calculated as follows:

- For $l \in [1,\ldots,q]$, $\boldsymbol{k_l}$ equals the number of non-zero values in the $l$-th column of W1.
- For each clause $C_l$ has a head $A_l$, the $y$-th neuron stands for the atom $A_l$. *Then* $\boldsymbol{\mu_l}$ equals the number of positive values in the $y$-th column of W2.
- Calculate $A_{min} > \frac{MAX_p(\vec{k},\vec{u})-1}{MAX_p(\vec{k},\vec{u})+1}$
- Calculate $W \geq \frac{2}{\beta} * \frac{ln\,(1+Amin)-ln\,(1-Amin)}{MAX_p(\vec{k},\vec{u})*(Amin-1)+Amin+1}$
- Calculate the weights between the input layer and the hidden layer- W1'
  W1'=W1*$\boldsymbol{W}$
- Calculate the weights between the hidden layer and the output layer –W2'
  W2'=W2*$\mathbf{W}$
- For thresholds $l \in [1,\ldots,q]$ in the hidden layer , calculate $\boldsymbol{\theta_l} = \frac{(1+A_{min})(k_l-1)}{2}\boldsymbol{W}$.
- For o $\in [1,\ldots,n]$ in the output layer, denote the number of positive values in the $o$-th column of W2 as $r_o$, calculate $\boldsymbol{\theta}_o = \frac{(1+A_{min})(1-r_o)}{2}\boldsymbol{W}$.

It is clearly seen that the model of a Tp-neural network and the model of a CILP-neural network have the same tuple (*n, q, W1, W2*) for a general logic program.

**A declarative language and a code editor**

The users would like to use a declarative language for Logic programming, which would better to be similar to the well-known logic programming. The following language $\mathcal{L}$ introduced in this report is very similar to the turbo-prolog language.

A general logic program consists of facts, rules and questions. In order to present facts, rules and questions well, a general logic program in $\mathcal{L}$ is designed to have four sections – Domains, Predicates, Goals, Rules.

In Domains section, we define terms and classify them into domains. In Predicates section, we define predicates which are used in logic clauses. In Goals section, we state questions which are supposed to be answered by machine. In Rules section, we present the facts and logic clauses. The deliberate grammar design of a general logic program in $\mathcal{L}$ is shown as follows:

1. Domains' Format:
   <domain name> = { <list of terms> }.
2. Predicates' Format:
   <predicate name> = ( <list of domain> ).
3. Goals' Format:
   ?<predicate name> ( <list of terms or variables> ).
4. Rules divide into facts and clauses.
   Facts' format:
   <predicate name> ( <list of terms> ).
   Clauses' format:
   <predicate name> ( <list of terms or variables> ) :-
   <predicate name> ( <list of terms or variables> )
   {;<predicate name> ( <list of terms or variables> ) }.

**Example 4.1.3.1** Considering the following program *P*:

```
Domains
  Suspect = {Harry,Jane,Stephen }.
Predicates
  HaveKeys ( Suspect ).
  HaveSmallFeet ( Suspect ).
  Smoke ( Suspect ) .
  Criminal( Suspect ) .
Goals
  Criminal(X).
Clauses
  HaveKeys(Harry).
  HaveKeys(Jane).
  HaveKeys(Stephen).
  HaveSmallFeet(Stephen).
  HaveSmallFeet(Jane).
  Smoke(Stephen).
Criminal(X):-HaveKeys(X);HaveSmallFeet(X);Smoke(X).
```

In *P* there are one domain, four predicates, one goal, six facts and one rules. Either a domain name or a predicate name is a string of laterals. A term is also a string of laterals; but it must start with a lower-case letter. A variable is a string of laterals which starts with an upper-case letter; and its scope is the current sentence in where it is. The design of a general logic program must obey the procedure – define domains, declare Predicates, set Goals, present Clauses (Facts and Rules), which is shown in the example above.

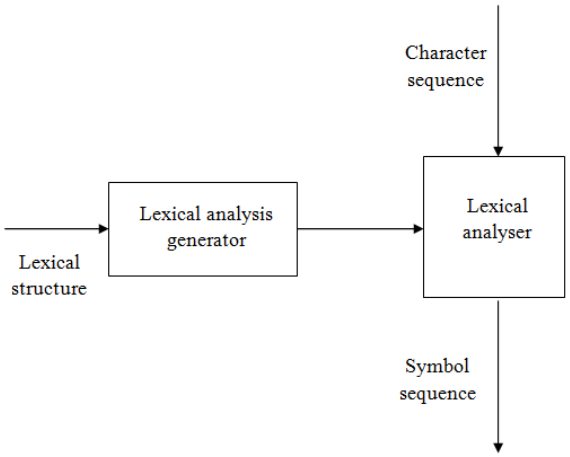
The code editor is a simple text editor, on which users write logic programs in $\mathcal{L}$.

**A translator**

A translator is a necessary component which connects the symbolic logic programming component to the connectionist computing systems. It is supposed to transform a general logic program written in $\mathcal{L}$ into a tuple (*n, q, W1, W2*) which either the model of a Tp-neural network or the model of a CILP-neural network could use to build a neural-symbolic system that can compute deduction.

The translator is very similar to a single pass compiler which consists of two stages. In the first stage, the translator performs lexical analysis, syntactic analysis and creates a token table. In the second stage, it performs intermediate representation generation and target tuple (*n, q, W1, W2*) creation.

*Lexical analysis* is a relatively simple phase in which symbols (or tokens) of the language are formed. It is supposed to indentify predicates, terms, variables, conjunctions.




The set of acceptable symbols is shown as follows:

1. A string –a sequence of laterals
2. '.'
3. '='
4. ":-"
5. "~"
6. '{' and '}'
7. '(' and ')'
8. ',' and ';'

*Syntax analysis is* a phase to in which the overall structure of a program is identified, and involves an understanding of the order in which the symbols in a program may appear. According the declarative language $\mathcal{L}$, there are four sections in a logic program. For each section, it has a label to mark it and there is a piece of syntax which is used to define its contents. The syntax for a general logic program in $\mathcal{L}$ is defined in a regular grammar as follows:

1. Labels/Key words: Domains, Predicates, Goals, Rules
   Its syntax is defined as follow:
   S -> Domains | Predicates | Goals | Rules
2. Domains
   Its syntax is defined as follow:
   S -> DOMAIN = { B }.
   B ->TERM
   T ->TERM, B
   DOMAIN -> [a-zA-Z]
   DOMAIN -> [a-zA-Z]，DOMAIN
   TERM -> [a-z] [a-zA-Z]*
3. Predicates
   Its syntax is defined as follow:
   S-> PREDICATE = ( B ) .
   B -> DOMAIN
   B -> DOMAIN, B
   PREDICATE -> [a-zA-Z] [a-zA-Z]*
   DOMAIN -> [a-zA-Z] [a-zA-Z]*
4. Goals
   Its syntax is defined as follow:
   S -> ? PREDICATE ( B ) .
   PREDICATE -> [a-zA-Z] [a-zA-Z]*
   B -> TERM
   B -> TERM, B
   B -> VARIABLE
   B -> VARIABLE, B
   TERM -> [a-z] [a-zA-Z]*
   VARIABLE -> [A-Z] [a-zA-Z]*
5. Facts
   Its syntax is defined as follow:
   S -> PREDICATE ( B ) .
   PREDICATE -> [a-zA-Z] [a-zA-Z]*
   B -> TERM
   B -> TERM, B
   B -> VARIABLE
   B -> VARIABLE, B
   TERM -> [a-z] [a-zA-Z]*
   VARIABLE -> [A-Z] [a-zA-Z]*

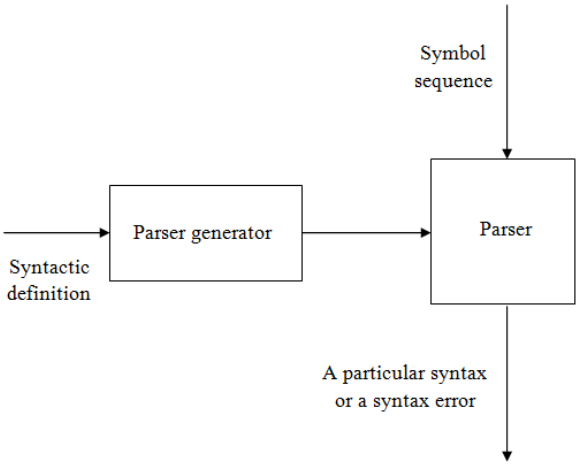
`

6. Clauses

   Its syntax is defined as follow:

   S -> A :- T .
   T -> A
   T -> ~A
   T -> A; T
   T -> ~A; T
   A -> PREDICATE ( B )
   PREDICATE -> [a-zA-Z] [a-zA-Z]*
   B -> TERM
   B -> TERM, B
   B -> VARIABLE
   B -> VARIABLE, B
   TERM -> [a-z] [a-zA-Z]*
   VARIABLE -> [A-Z] [a-zA-Z]*

According these grammars, the syntax analyser (or parser) can fit a sequence of tokens into a specified syntax. A parsing problem consists of finding a derivation (if one exists) of a particular sentence using a given grammar. In this translator, the parser is a left to right bottom-up parser with one symbol lookahead LR (1). The LR (1) is a very common algorithm in complier design, which will not be introduced here, see [7, 11].




*Intermediate representation generation* is a phase to transform each statement in the original code of a general logic program to intermediate data according to a particular syntax.

Given a general logic program P, the translator is supposed to generate appropriate intermediate data which it can use to create the target tuple to construct a neural-symbolic network representing P. Four types of intermediate data are defined as follows:

(1) *n*, the number of neurons of the input layer or the output layer;
(2) *q*, the number of neurons of the hidden layer;
(3) *Input-hidden-connection :(*$input_i$ *,* $hidden_j$*,* $weight_{ij}$*)*, a connection between a neuron in the input layer and a neuron in the hidden layer with a weight.
(4) *Hidden-output-connection:(*$hidden_j$*,*$output_k$*,*$weight_{jk}$*)*, a connection between a neuron in the hidden layer and a neuron in the output layer weight.

When the translator identifies the syntax of a statement, it will either create a new item of intermediate data or amend the value of a previous item of intermediate data. When a domain statement or a predicate statement is encountered, n will be amended. When a fact or a clause is encountered, q will be amended; and input hidden connections will be created.

*Target tuple creation is* a simple phase to create a tuple (*n, q, W1, W2*) according to the intermediate data. n is assigned the final value of n in the intermediate data. q is also assigned the final value of q in the intermediate data. W1 is a *n*-by-*q* matrix which is initialised with 0s. For each i*nput-hidden-connection :(*$input_i$*,* $hidden_j$*,* $weight_{ij}$*), the* entry *W1 [*$input_i$ *,* $hidden_j$*]* is assigned $weight_{ij}$. W2 is a *q*-by-*n* matrix which is initialised with 0s. For each *Hidden-output-connection :(* $hidden_j$*,*$output_k$*,* $weight_{jk}$*), the* entry *W2[* $hidden_j$*,* $output_k$*]* is assigned $weight_{jk}$.

The final task for the translator is to with set up a neural-symbolic system with the tuple(*n, q, W1, W2*). The neural-symbolic system could be either a Tp-neural network or a CILP neural network, which depends on the choice of users.

**An interpreter and a result reader**

An interpreter is a component that interprets the execution result of a neural-symbolic system into propositional variables' truth value and gives symbolic answers to the goals of P.

According the record of the meaning (model) of the neurons in the output layer, it is easy to judge a propositional variable' truth value. For a Tp-neural network, if the value of a neuron is greater than or equal to 0.5, then its corresponding propositional variable is considered to own a true value; otherwise the propositional variable owns a false value. For a CILP-neural network, if the value of a neuron is greater than or equal to $A_{min}$, then its corresponding propositional variable is considered to own a true value.

The interpreter is supposed to understand the meaning of goals in a logic program, search answers and then format answers in a symbolic way. There are two types of goals. One has no variable and the other one has. A goal with no variable expects an answer whether the corresponding proposition variable is true or not. A goal with variables expects an answer that presents the sets of variables' value that make the goal's propositional variable true.

A result reader is a component that either presents the interpreter's answers to goals in P or states all errors when P has grammar errors.

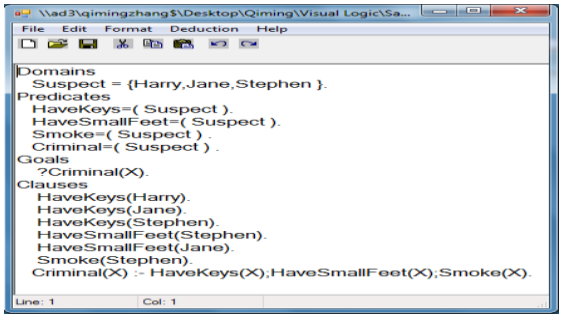
*Implementation*

The Logic Programming Interface is implemented in C#. It consists of three class libraries and one windows application shown in detail as follows:

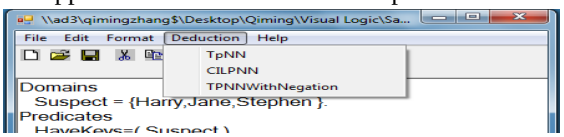
1. Linear-Algebra-Library Class Library, which provides RVector and RMatrix structures with proper algebra operations, see [10].
2. Tp-Neural-Network Class Library, which provides a Tp-Neural-Network class with the following methods:
   a) Tp(leteralsNum,clauseNum,W1,W2), which is the constructer;
   b) Generate(), which is a method that generates a entire Tp-neural network, which includes neurons, weights and thresholds;
   c) TpFixedPoint(), which is a method that a Tp-neural network iterate itself doing massively parallel computing to obtain the Fixed Point.
3. CILP-Neural-Network Class Library, which provides a CILP-Neural-Network class with the following methods:
   a) CILP(leteralsNum,clauseNum,W1,W2), which is the constructer;
   b) Generate(), which is a method that generates a entire CILP-neural network, which includes neurons, weights, threasholds, *Amin* and *W*;
   c) CILPFixedPoint(), which is a method that a CILP-neural network iterate itself doing massively parallel computing to obtain the Fixed Point.
4. Logic-Interface Windows Application, which provides a code editor, a translator , an interpreter and a result reader.

**Example 5.1.1** Consider the logic program in Example 2.1.1. A simple procedure to use application shown as follows:

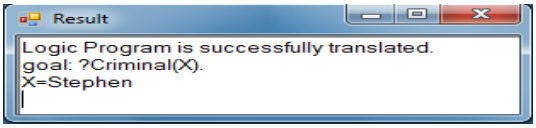
1. Type in a logic program P which accords to the grammar of the language $\mathcal{L}$.




2. Choose a neural-symbolic system model from TpNN and CILPNN, both of which can do deduction.Then the application translates P and set ups the model.




3. The neural symbolic system iterates massively parallel computing and obtains the fixed pointed. The interpreter will generate answers to Goals in P. The result reader shows the result as follows.




## 4 A knowledge refining system

*Specification*

Knowledge refining is to insert background knowledge (or coarse knowledge) of a particular domain into a neural network and obtain fine knowledge by learning with example data. One kind of Neural-symbolic systems – CILP is very suitable to do knowledge refining. Not only CILP has the capability to present Background Knowledge into neural networks, but also it can use back-propagation to get networks trained with examples.

Using the interface above to translate a logic program to a CILP-neural network can be considered as the first step of knowledge refining. The second step is to deploy standard back-propagation to train the CILP-neural network.

Using the interface to do knowledge refining has two merits:

1. An easy way to generate neurocomputing models. We use a logic programming language to build a neural network with background knowledge. The logic programming language is syntactically similar to the way people reason, which makes for a general and easily accessible interface for users with diverse backgrounds to do knowledge refining.
2. Neural networks will incorporate rules rather than eliminate them when trained with examples. During the process of training, the embedded knowledge in a CILP-neural network is refined and coarse knowledge becomes fine knowledge.

*Design*

Consider the neural network software which the police officer has, it is could be the outcome of a knowledge refining system. In this report, it will show how to obtain such a system.

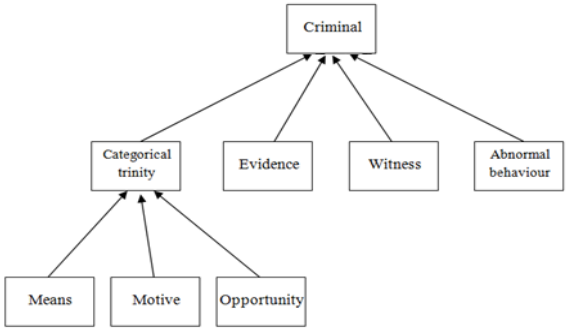
First, we will introduce some basic knowledge about real crime analysis and a model to solve crime detective problems. Then we will show how to present the knowledge in a logic programming language. Finally we obtain a CILP neural network which could be trained with examples, and we how to train it.

**Crime detection knowledge**

Some terminology in crime detection:

1. The categorical trinity—means, motive, opportunity. Respectively, they refer to: the ability of the suspect to commit the crime (means), the reason the suspect had to commit the crime (motive), and whether or not the suspect had the chance to commit the crime (opportunity), see [13].
2. Evidence. Evidence may be left on the scene after criminals have committed a crime. Some people may see the crime and become a witness.
3. Witness. A witness is someone who has firsthand knowledge about a crime through his or her senses (e.g. seeing, hearing, and smelling).
4. Abnormal behaviour. Criminals tend to have abnormal behaviour after the crime. The most common one is lying.

As shown above, six elements have to be taken into consideration to judge who criminals are. They are Means, Motive, Opportunity, Evidence, Witness, and Abnormal Behaviour. The following graph shows their attribution to determine a criminal in a clear way.




**Logic programming representation**

Instead of using the decision architecture above, the police officer may have the following logic when deciding who criminals are:

1. Categorical-trinity(x) <- Means(x), Motive(x), Opportunity(x)
2. Criminal(x) <- Categorical-trinity(x), Evidence(x), Abnormal-behaviour(x)
3. Criminal(x) <- Categorical-trinity(x),Witness(x), Abnormal-behaviour(x)
4. Criminal(x) <- Evidence(x), Witness(x), Abnormal-behaviour(x)

With such logic, a CILP neural network can be obtained. However, such a network obviously cannot help the police officer to find suspects and rule out the suspension of unrelated people. Things are complicated and uncertain in real crime scenarios. The police officer wants the problem-solving assistant to help me make judgements with uncertain information or incomplete information.

**Knowledge refining**

According to Theory of Uncertain Reasoning, there is a notion called the credibility, which defines the extent to which one person believes one thing or one event due to experience. We borrow this notion to CILP neural networks. For each neuron in the input layer and the output layer, it has a domain ranging from -1 to 1. If we define the credibility $I \in [-1, 1]$ for each neuron, then we can interpreter neurons in the CILP neural network as follows:

- If I is equal to -1, it is certain that the model of the neuron does not exist.
- If I is equal to 1, it is certain that the model of the neuron does exist.
- If I is equal to 1, it implies that no evidence proves the existence of the model of the neuron.
- If $I \in (0, 1)$, the model of the neuron has a probability of existence.
- If $I \in (-1, 0)$, the model of the neuron has a probability of nonexistence.

If we have sufficient examples to train the CILP neural network, the deduction knowledge can be refined so that the police officer can use the neural network software to do uncertain reasoning. This illustration gives a design of a knowledge refining system which can assist police officer to solve crime problems. However, in terms of the fact that this is an undergraduate honours project, our design is not implemented because no example data are available. It can be regarded as a methodology to use a neural-symbolic system to do knowledge refining.

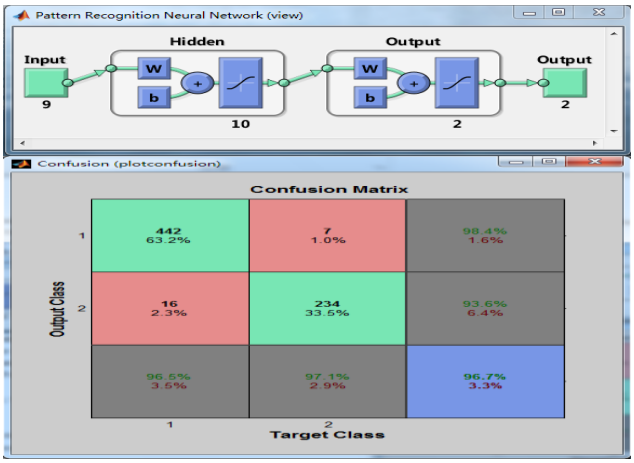
*Implementation*

In this report, we conduct a test by using the cancer data set from the UCI Machine Learning Repository, see [14]. These data consisted of 9 clinical parameters (clump thickness, uniformity of cell size, uniformity of cell shape, marginal adhesion, single epithelial cell size, bare nuclei, bland chromatin, normal nucleoli, and mitoses) and 699 examples. Sixteen samples of this data set are including uncompleted data set. We used remained 683 examples in which 444 are benignant examples and 239 are malignant examples. The detailed information could be found in the appendix, see Figure 1 – Figure 9.

All nine parameters may be important risk factors for development of breast cancer. Inspired by [15], we acquired the coarse knowledge about the relationship

between these factors and breast cancer by using the machine learning method, then the knowledge was mapped to the structure of a neural network which is a CILP neural network. Then the CILP neural network was trained with the data set and got a better prediction. The operations are shown as follows:

1) We used Matlab to build a 3-layer neural network and trained it with the data set.




2) We deploy the pedagogical approach to get the coarse knowledge as follows:

i. For 9 input neurons, we simulate $2^9$ input data sets where each of 9 neurons either has a value of 0.1 or 1.

ii. From the result, we conclude the coarse knowledge as follows:

a) If the value of Bare Nuclei is large, then it is Malignant.

b) If the value of Clump Thickness is large and the value of one attribute from Uniformity of Cell Size, Uniformity of Cell Shape,Marginal Adhesion, Single Epithelial Cell Size, Bland Chomatin, Normal Nucleoli and Mitoses is large, then it is Malignant.

c) If the value of Uniformity of Cell Size is large and the value of one attribute from Bland Chomatin, Normal Nucleoli and Mitoses is large, then it is Malignant.

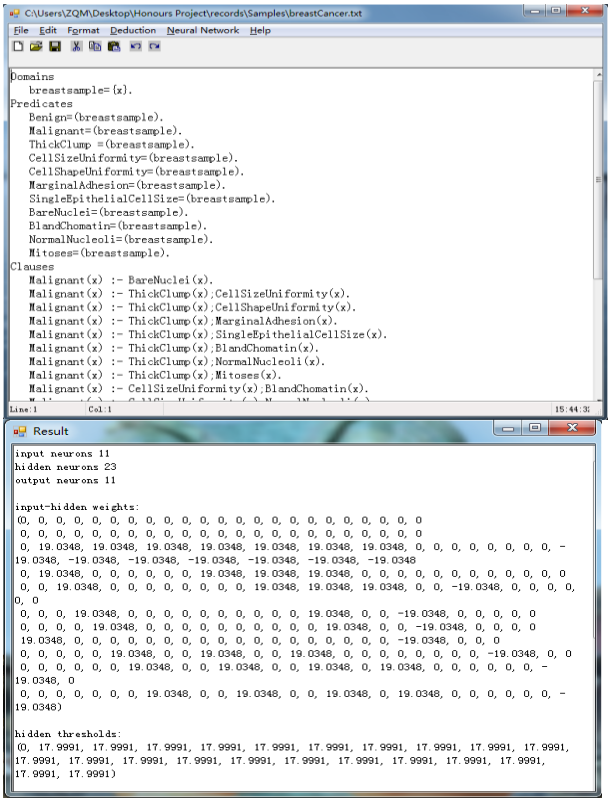
d) If the value of Uniformity of Cell Shape is large and the value of one attribute from Bland Chomatin, Normal Nucleoli and Mitoses is large, then it is Malignant.

e) If the value of Marginal Adhesion and the value of Normal Nucleoli are both large, then it is Malignant.

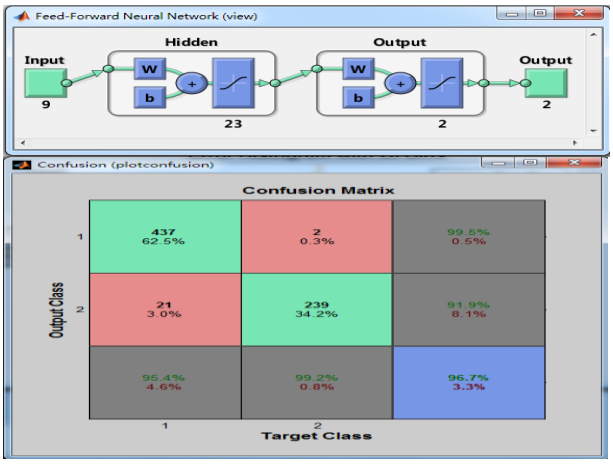
f) If the value of Single Epithelial Cell Size and the value of Mitoses are both large, then it is Malignant.

g) If the value of Clump Thickness is small and the value of one attribute from Uniformity of Cell Shape, Marginal Adhesion, Single Epithelial Cell Size, Bare Nuclei, Bland Chomatin, Normal Nucleoli and Mitoses is small, then it is Benign.

3) According to the coarse knowledge, we use the interface to generate the CILP neural network.




4) We copy the CILP neural network to Matlab and get trained with the cancer data set.




5) The test outcome is satisfying because the trained CILP has achieved the similar correctness as the pure machine method does. Moreover, the trained CILP neural network has a neat structure; and it can be interpreted to the symbolic knowledge. According to Figure 10 and Figure 11 in the appendix, all the rules above in the 2nd item hold. What is interesting is that

`

the performance of the CILP neural network cannot be improved by setting a better training goal while a general neural network can. This implies the knowledge embedded in the CILP neural network does not accept some data (to be exact, 23data).

# 5 Einstein's Riddle

*Specification*

The Einstein's Riddle is world-famous for its hardness. It is also said that 98% of the world population couldn't find a solution in the late 1800s. Such a difficult problem can arise in real crime solving. Imagine that, in Example 2.1.1 the police found the cigarette was Dunhill's and the feet prints led to a street exactly like the street in Einstein's Riddle below. The goal is to find in which of the houses the offenders lives.

There are many efficient techniques to solve this question, for example, constraint programming. In this report, we give a model in logic programming to solve this riddle. The following is one version of the Einstein's Riddle:

- There are five houses in a row, each of different colours.
- Each has an owner of a different nationality.
- Each owner has a unique favourite drink, type of cigarette, and a pet.
- The question is: Who owns the fish?
- Necessary clues**:**
  1. The Englishman lives in the red house.
  2. The Swede keeps dogs.
  3. The Dane drinks tea.
  4. The green house is just to the left of the white one.
  5. The owner of the green house drinks coffee.
  6. The Pall Mall smoker keeps birds.
  7. The owner of the yellow house smokes Dunhills.
  8. The man in the centre house drinks milk.
  9. The Norwegian lives in the first house.
  10. The Blend smoker has a neighbour who keeps cats.
  11. The man who smokes Blue Masters drinks bier.
  12. The man who keeps horses lives next to the Dunhill smoker.
  13. The German smokes Prince.
  14. The Norwegian lives next to the blue house.
  15. The Blend smoker has a neighbour who drinks water.

*Design*

According the riddle above, we can get three types of information as follows:

1. Direct information from clues;
2. Indirect information from clues;
3. Constraint information.

**Example 4.3.1** Consider clue 1- "The Englishman lives in the red house."

1. Two pieces of direct information: (a) If the Englishman lives in House N, then House N are red. (b) If House N is red, then the Englishman lives in House N.
2. Eight pieces of indirect information: (a) If a man (not English, can be Norwegian, Dane, German or Swedish) live in House N, and then House N is not red. (b) If House N is one colour form Yellow, Blue, Green and White, then the English doesn't live in House N.
3. Four pieces of constraint information: (a) If House N is none of Yellow, Blue, Green and White, and then House N is red. (b) If none of the Norwegian, Dane, German or Swedish lives in House N, then the English lives in House N.(c) If House N is Red, then House N is none of Yellow, Blue, Green and White. (d) If the English lives in House N, then none of the Norwegian, Dane, German or Swedish lives in House N.

The information needs to be reformed in a definite logic program. In order to express the presence of negation, we need to use extended logic programs, which has been introduced in Definition 3.4.1, Definition 3.4.2 and Theorem 3.4.1. In the following, we introduce a solution to solve Einstein's Riddle in the positive form $P^*$ of an extended logic program $P$ with using the logic programming interface. We define domains and predicates for $P^*$ as follows:

```
Domains
  HouseNumber={1,2,3,4,5}.
  Color={Yellow,Blue,Red,Green,White}.
Nationality={Norwegian,Dane,English,German,Swede}.
Drink={Water,Tea,Milk,Coffee,Beer}.
Cigarette={Dunhill,Blend,PallMall,Prince,BlueMaster}.
  Pets={Cats,Horses,Birds,Fish,Dogs}.
Predicates
  HouseColor(HouseNumber,Color).
  NegatedHouseColor(HouseNumber,Color).
  OwnerIs(HouseNumber,Nationality).
  NegatedOwnerIs(HouseNumber,Nationality).
  OwnerDrink(HouseNumber,Drink).
  NegatedOwnerDrink(HouseNumber,Drink).
  OwnerSmoke(HouseNumber,Cigarette).
  NegatedOwnerSmoke(HouseNumber,Cigarette).
  OwnerPet(HouseNumber,Pets).
  NegatedOwnerPet(HouseNumber,Pets).
```

HouseColor(1,Red) is an atom that House 1 is Red. NegatedHouseColor(1, Red) is also an atom that House 1 is not Red. NegatedHouseColor is the negation of

`

HouseColor, which expresses the negation in an explicit way. For each of the predicates, there is a positive form with a prefix - negated to express the negation of the predicate.

**Example 4.3.2** Consider Example 4.3.1. The information is reformed as follows:
Direct information:
(a) OwnerIs(X,English):-HouseColor(X,Red).
(b) HouseColor(X,Red):-OwnerIs(X,English).
Indirect information:
(a)NegatedHouseColor(X,Red):-wnerIs(X,Norwegian).
(b)NegatedHouseColor(X,Red):-OwnerIs(X,Dane).
(c)NegatedHouseColor(X,Red):-OwnerIs(X,German).
(d)NegatedHouseColor(X,Red):-OwnerIs(X,Swede).
(e)NegatedOwnerIs(X,English):HouseColor(X,Yellow).
(f)NegatedOwnerIs(X,English):-HouseColor(X,Blue).
(g)NegatedOwnerIs(X,English):-HouseColor(X,Green).
(h)NegatedOwnerIs(X,English):-HouseColor(X,White).
Constraint information:
(a) HouseColor(X,Red):-
NegatedHouseColor(X, Yellow);
NegatedHouseColor(X, Blue);
NegatedHouseColor(X, Green);
NegatedHouseColor(X, White);
(b)OwnerIs(X, English):-
NegatedOwnerIs(X, Norwegian);
NegatedOwnerIs(X, Dane);
NegatedOwnerIs(X, German);
NegatedOwnerIs(X, Swede).
(c)NegatedHouseColor(X, Yellow):-
HouseColor(X, Red).
NegatedHouseColor(X, Blue):-HouseColor(X, Red).
NegatedHouseColor(X, Green):-HouseColor(X, Red).
NegatedHouseColor(X, White):-HouseColor(X, Red).
(d)NegatedOwnerIs(X, Norwegian):-
OwnerIs(X, English).
NegatedOwnerIs(X, Dane):-
OwnerIs(X, English).
NegatedOwnerIs(X, German):-
OwnerIs(X, English).
NegatedOwnerIs(X, Swede):-
OwnerIs(X, English).

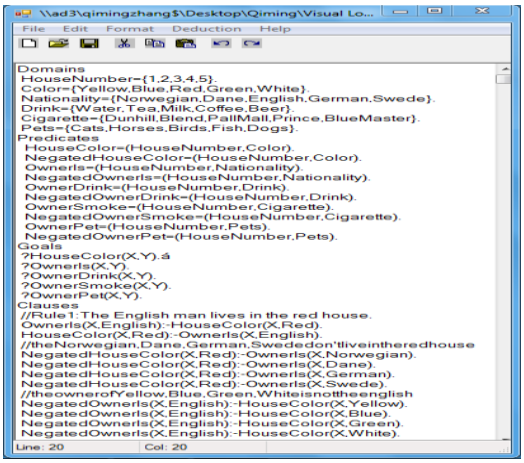
In this solution, we need find direct information and indirect information for all clues and constraint information for every atom, then reform the information in a logic program. Then we can use neither a Tp-neural network nor a CILP system to compute the answer set.

Discussion about this solution:

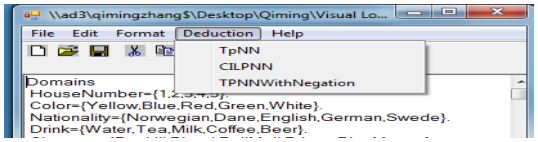
1. An extended program. In order to solve this question in logic programming, we need a way to express the negation explicitly. The extended program can do that. However, there is no logic programming tool that we can deploy in practice to use an extended logic program to do deduction. We resort to the positive form of an extended logic program. Consequently inconsistency may happen if the logic program is not well-behaved. Fortunately the result of this solution is a well-behaved logic program.
2. Comparison with Constraint Programming. Constraint Programming is based on Constraint propagation and Backtracking while neural-symbolic systems are based on Fixed Pointed Theorem. Therefore the search for a solution is trivial while the solving process of logic programming is monotonic.

*Implementation*

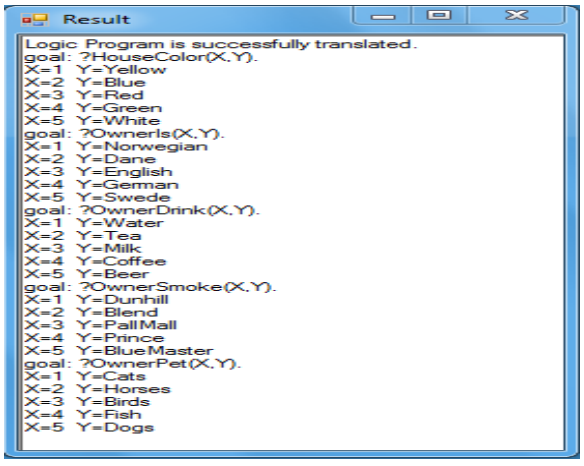
Solving the Einstein's Riddle in practice involves the follow procedures:

1. Type in the logic program with classical negation.
2. 

3. Choose a neural-symbolic system model from TpNN and CILPNN, both of which can solve this logic program.




4. The result is show as follows:



By comparing the execution time (0.817592s on averages) of this logic program and the execution (0.000000 on averages) of a constraint program using the Minion solver, the solver is quicker. However, the result doesn't mean that monotonic deduction is worse than repeated search with constraint propagation.

## 6 An Experiment on Classical Negation

*Specification*

In this section, we propose a new model of neural-symbolic networks to enable classical negation.

[1] "According to Lifschitz and McCarthy, commonsense knowledge can be represented more easily when classical negation (¬), sometimes called explicit negation, is available. In [12], Gelfond and Lifschitz have extended the notion of stable models to programs with classical negation."

[1] General logic programs provide negative information implicitly, by the closed-world assumption, while extended programs include explicit negation, allowing the presence of incomplete information in the data base. In the language of extended programs, we can distinguish between a query which fails in the sense that it does not succeed, and a query which fails in stronger sense that its negation succeeds.

**Example** [1,12] "Consider the following problem, due to John McCarthy, illustrate such a difference: a school bus may cross railway tracks unless there is an approaching train. This would be expressed in a general logic program by the rule cross ← ~train, in which case the absence of train in the database is interpreted as the absence of an approaching train, i.e. using the closed-world assumption. Such an assumption is unacceptable if one reasons with incomplete information. However, if we use classical negation and represent the above knowledge as the extended program: cross ← ¬train, the cross will be derived not be derived until the fact ¬train is added to the database."

**Definition 3.4.1** [12] An extended logic program is a finite set of clauses of the form $L_0 \leftarrow L_1, \ldots, L_m, \sim L_{m+1}, \ldots, \sim L_n$, where $L_i$ $(0 \leq i \leq n)$ is literal ( an atom or the classical negation of an atom, denoted by ¬) and ~ is default negation ( or negation-as-failure).

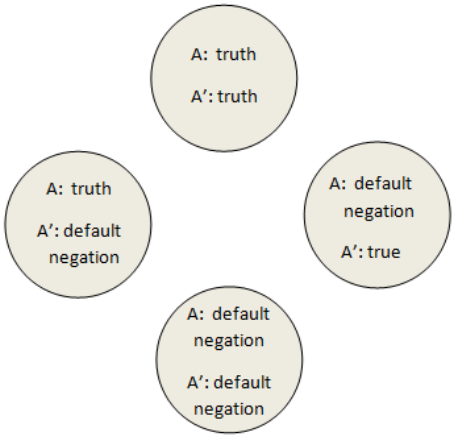
It is clearly seen that an atom A has four states:

1. A, positive existence
2. ¬A, classical negative existence
3. ~A , default negation of positive existence
4. ~¬A, default negation of classical negative existence.

In [1], Artur S. D'Avila Garcez, Krysia Broda and Dov M. Gabbay extended CILP to incorporate classical negation by using the positive form of an extended logic program.

**Definition 3.4.2** [1]The positive form (P*) of an extended logic program (P) is defined as follows: For any negative literal ¬A occurring in P, let A' be a positive literal form of ¬A. P* is obtained from P by replacing all the negative literals of each rule of P by its positive form.
**Note**: It can be clearly seen that P* is just an ordinary definite logic program.

In the positive form of an extended logic program, the original atom A becomes two atoms A and ¬A with four states as follows:




**Theorem 3.4.1** [1] For each extended logic program *P*, there exists a feedforward artificial network N with exactly one hidden layer and semilinear neurons such that *N* computes $T_{P*}$, where *P** is the positive form of *P*.
**Note**: Given a extended logic program P, there also exists a Tp-neural network which can compute $T_{P*}$, where *P** is the positive form of *P*.

By far, no matter it is a general logic program or an extended logic program, each of the atoms has only two states – true and default negation. In other words, each of the neurons in the input layer or the output layer of its corresponding neural-symbolic network also has only two states. However a neuron with real number value can have as many states as possible. Why do we need two neurons to present the three states – truth, negation, default (or state unknown) of an atom in an extended logic program? Is there an alternative to incorporate classical negation into neural-symbolic systems?

An experiment that tries to build a new model of neural-symbolic systems which incorporates classical negation is to change the model of Tp-neural networks so that it can do deduction with classical negation. This new model is supposed to meet the following requirements:

1. It has to incorporate classical negation into logic programs.

`

2. It should provide a translation algorithm to translate a logic program with classical negation to a connectionist system, which can compute *Tp*.
3. It should provide an algorithm such that the corresponding connectionist system of a logic program with classical negation can perform massively parallel deduction.

*Design*

In this report, we give a logic program with classical negation, its semantics, a Translation Algorithm to a symbolic-neural system and a Massively Parallel Deduction Algorithm.

**Definition 4.4.1** A logic program with classical negation is a finite set of clauses of the form $L_0 \leftarrow A_1, \cdots, A_m, \neg A_{m+1}, \cdots, \neg A_n$ $(0 \leq m \leq n)$, where $L_0$ is a literal (an atom $A_0$ or the classical negation of an atom $\neg A_0$) and $A_i$ denote an atom.

In such a logic program P, an atom A has three states – truth, classical negation, default (or state unknown). For a clause $L_0 \leftarrow A_1, \cdots, A_m, \neg A_{m+1}, \cdots, \neg A_n$ $(0 \leq m \leq n)$, $L_0$ will be concluded only if $A_i$ $(1 \leq i \leq m)$ has a truth state and $A_i$ $(m+1 \leq i \leq n)$ has a classical negation state; otherwise the atom $A_0$ will have a default state.

A logic program with classical negation has a different semantics from the semantics of a definite logic program. In the following we borrow notions from extended programs to explain the semantics of a logic program with classic negation.

[12] Let *Lit* be the set of ground literals of a logic program with classical negation. The Answer Set is a smallest subset *S* of Lit such that

1. for any rule $L_0 \leftarrow L_1, \ldots, L_m$, if $L_1, \ldots, L_m \in S$, then $L_0 \in S$;
2. if *S* contains a pair of complementary literals (e.g. A, ¬A), then *S* = *Lit.*

A well-behaved logic program with classical negation has exactly one answer set, and the set is consistent. The answer that a logic program with classical negation returns for a ground query (A) is yes, no, or unknown, depending on whether its answer set contains A, ¬A or neither.

Let P be a logic program with classical negation. Let *B* be the set of all grounded atoms and their classical negations determined by P. The collection of the states of all the atoms in *B* is an interpretation, denoted as *I*. Then $2^B$ is the set of all possible interpretations.

**Definition 4.4.2** Let P be a logic program with classical negation. The mapping $Tp: 2^B \rightarrow 2^B$ is defined as follows.

Let *I* be an interpretation of *B*; then $Tp(I) = \{L_0 \mid L_0 \leftarrow L_1, \ldots, L_n$ *is a clause in P* and $\{L_1, \ldots, L_n\} \subset I\}$.

**Proposition 4.4.1** Let P be a well-behaved logic program with classical negation. The answer set $S = Tp \uparrow w$.

For a logic program with classical negation P, there exists a three-layer feedforward artificial network N such that *N* computes $T_P$. the following is the translation algorithm.

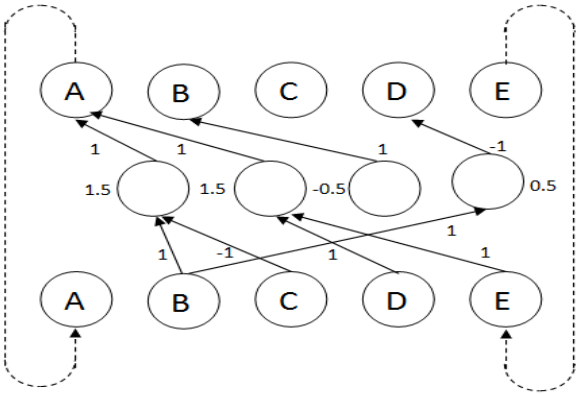
**Translation Algorithm**

Let p and q be the number of propositional variables and the number of clauses occurring in P, respectively. Without loss of generality we may assume that the grounded atoms are numbered from 1 to p. The network associated with P can now be constructed as follows:

1) The input and output layer is a vector of p neurons, where the *i*th neuron represents the atom $A_i$, $1 \leq \mathbf{i} \leq p$. The threshold of each neuron occurring in the input or output layer is set to 0.
   The activation function in the hidden layer is the linear limit function (y = x).
   The activation function in the hidden layer is the hard limit function ($y = \begin{cases} \mathbf{1}, & \mathbf{if\ x \geq 0.5} \\ \mathbf{0}, & \mathbf{others} \end{cases}$).
   The activation function in the output layer is the function ($y = \begin{cases} \mathbf{1}, & \boldsymbol{if\ x \geq 0.5} \\ \mathbf{-1}, & \boldsymbol{if\ x \leq -0.5} \\ \mathbf{0}, & \boldsymbol{others} \end{cases}$).
2) For each clause of the form $L_0 \leftarrow A_1, \cdots, \mathbf{A_m}, \neg A_{m+1}, \cdots, \neg A_n$ $(0 \leq \mathbf{m} \leq n)$,occurring in P do the following:
   a) Add a binary threshold unit c to the hidden layer of N.
   b) Connect c to the unit representing $L_0$ in the output layer. If $L_0$ is negated atom, set up the connection with weight -1; otherwise set up the weight 1
   c) For each $A_i$, $1 \leq \mathbf{i} \leq m$, connect the neuron representing $A_i$ in the input layer of N to c, and set the connection's weight to 1.
   d) For each $A_i$, $m+1 \leq \mathbf{i} \leq n$, connect the neuron representing $A_i$ in the input layer of N to c, and set the connection's weight to -1.
   e) Set the threshold $\boldsymbol{\theta_c}$ of c to n- 0.5.

**Example 4.4.1**Consider the following logic program P:

1) A ←B, ¬C
2) A ← D,E
3) B ←
4) ¬D ←B




The above neural network N is obtained by applying the above algorithm to P. If B = 1 and C =-1 in the input layer of N then A = 1 in the output layer of N, representing the rule A ← B ¬ C of P. Similarly, B = 1 is always obtained in the output layer of N, regardless of its input vector, indicating that B is a fact. D = -1 is always obtained because B is a fact and the rule ¬D ← B holds.

**Proposition 4.4.1** Let P be a well-behaved program. There exists a single hidden layer recurrent network such that each computation starting with an arbitrary initial input I converges to a stable state and yields the unique fixpoint of Tp.

The following is the algorithm to compute the stable state of P in a neural network mentioned above.

**Massively Parallel Deduction Algorithm**

Let p be the number of input neurons and the number of output neurons in Tp-neural network N. The input is defined by a vector $I = (I_1, \ldots, I_p)$ and the output is given by a vector $O = (O_1, \ldots, O_p)$.

1. Initialize I = [0, 0, …, 0].
2. Loop:
   4) Calculate O = feedforward(I);
   5) If I is equal to O, then terminate;
   6) If I is not equal to O, then for j in (1, …, p), replace the value of $I_j$ with the value of $O_j$ in O.

**Example 4.4.2** Consider the logic program in Example 4.4.1.As shown in Figure, N is recurrently connected, its output vector feeds the input vector in an iteration of Tp. Let the initial input vector I = (0, 0, 0, 0, 0). So the deduction process would be as follows:

*(5)* $Tp \uparrow 0 = \{0, 0, 0, 0, 0\}$;
*(6)* $Tp \uparrow 1 = \{0, 1, 0, 0, 0\}$;
(7) $Tp \uparrow 2 = \{1, 1, 0, 0, -1\}$;
*(8)* $Tp \uparrow 2 = Tp \uparrow 3 = Tp \uparrow w$ .

Discussions about this model:

- Answer Set: Herbrand model does not work because each atom has three states. We need a way to represent a logic program in a set of grounded instances (or literals).
- Inconsistency issue: Inconsistency may happen when the logic program is not well-behaved. Solutions can be borrowed from the positive form of extended programs, see [1].
- "¬Q ← not P" failure, see [12]. "¬Q ← not P" means: Q has a classical negation state, if P has a default state. But this model cannot do such deduction.
- The translation algorithm is not well explained. For clause $\neg A_0 \leftarrow A_1, \cdots, A_m, \neg A_{m+1}, \cdots, \neg A_n$ $(0 \leq m \leq n)$, a neuron c will be added to the hidden layer. The connection from c the $A_0$ is weighted as -1. Consider the clause ¬D ←B, a neuron c is added to the hidden layer to represent this clause, the connection between c and the neuron which represent D is -1 which is easy to interpret it as D ←¬B. However, if -1 can be taken as an inhibitory signal, we might give an interpretation to ¬D ←B as B will transmit an inhibitory signal to D and make D in a state of inhibition if B receives an excitatory signal.
- Merit one: The model can perform a certain type of deduction with classical negation.
- Merit two: The model gives a means to use one neuron to present the three states of one atom instead of two neurons in the model of the positive form of extended programs.
- CILP system can also be changed to incorporate classical negation in a similar way.

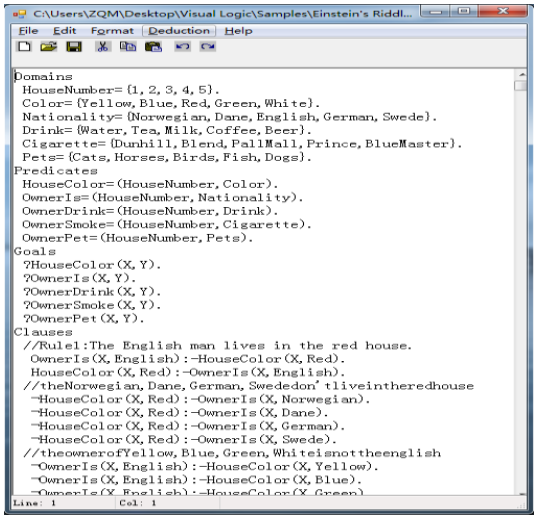
*Implementation*

We implemented this new model of neural-symbolic networks in the logic programming interface. It required a library class, representing the model, which consists of a constructor, Generate() and TpFixedPoint(). Another two new components - a new translator and a new interpreter are needed to be added to the Logic-Interface Windows Application.

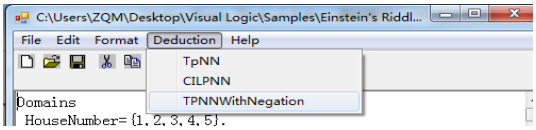
**Example 5.4.1** The following is the procedure to use the model to solve Einstein's Riddle.

1. Type in the logic program with classical negation.

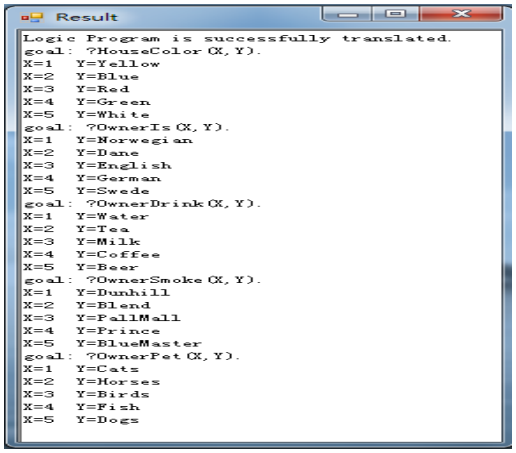
`




2. Choose the neural-symbolic system – TPNNWithNegation.




3. The result is shown as follows:




This solution needs 2104 neurons and 463500 weights while the solution using the positive form of an extended program needs 2354 neurons and 927000 weights. Both solutions need 14 iterations to arrive at the fixed point. Definitely it will save a lot of memory and execution time.

## 7 Conclusion

A logic programming interface which deploys neural-symbolic systems to do deduction been proposed in this report. Based on it, we proposed a novel approach to build knowledge refining systems and tested it with the breast cancer problem. Then we tested the soundness of the interface by solving the Einstein's Riddle. Finally, we proposed a new type of neural-symbolic systems which enables classical negation. Future work will focus on a deep research on knowledge refining systems and a better solution to build a neural-symbolic system which enables classical negation.


## Acknowledgments

I owe sincere and earnest thankfulness to Dr Ekaterina Komendantskaya who is my supervisor during this project. She gave me a lot of motivation to do my best to do my project. It is always her who got me in the correct way before I went further in the wrong direction. I also would like to thank my friend Huo Shuaipeng for encouraging me to work hard on this project all the time.


## References


[1] Artur S. D'Avila Garcez, Krysia Broda, Dov M. Gabbay. Neural-Symbolic Learning Systems[M]. Berlin: Springer. 2002

[2] S. Holldobler, Y. Kalinke, Towards a massively parallelcomputational model for logic programming[J], in:Proc. ECAI-94 Workshop on Combining Symbolic and Connectionist Processing, 1994

[3] Ekaterina Komendantskaya. Learning and Deducion in Neural Networks and Logic[D], Ireland: Department of Mathematics, University College, Cork (UCC), 2007

[4] Andries P. Engelbrecht. Computational Intelligence: An Introduction [M], John Wiley, New York,2003.

[5] Artur S. D'Avila Garcez, Lu ́s C. Lamb, Dov M. Gabbay. Neural-symbolic cognitive reasoning[M], Berlin: Springer. 2009

[6] Sebastian Bader and Pascal Hitzler. Integrating Logic Programs and Connectionist Systems - Introductory Course at ESSLLI 2008. Hamburg, Germany, August 2008. From http://www.neural-symbolic.org/

[7] Robin Hunter. THE ESSENCE OF COMPILERS[M]. London: Prentice Hall, 1999

[8] Nancy Ide. CMPU 331: Compiler Design. Spring 2009. From http://www.cs.vassar.edu/~cs331/

[9] Richard O. Duda, Peter E. Hart, David G. Stork, Pattern Classification, 2nd Edition, Wiley, 2000, ISBN 978-0-471-05669-0

[10] Dos Passos Waldemar. Numerical methods, algorithms, and tools in C♯.CRC Press, 2009,ISBN: 0849374790

[11] Dick Grune, Ceriel J. H. Jacobs, Koen G. Langendoen.Modern Compiler Design. John Wiley, New York,2000

[12] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. New Generation Computing, 9:365-385,1991.

[13] *See* Wikipedia, *Means, motive, and opportunity*. http://en.wikipedia.org/wiki/Means,_motive,_and_opportunity (as of 11th April, 2011, 22:50 GMT).

[14] Murphy,P.M., Aha, D.W. (1994). UCI Repository of machine learning databases. Irvine, CA: University of California, Department of Information and Computer Science.
From http://www.ics.uci.edu/~mlearn/MLRepository.html

[15] Jicheng Wang. A Knowledge Acquisition System Based on Symbolic Neural Network. Acta Electronica Sinica, Vol 26 No 8. Aug. 1998.

`

**Appendices**

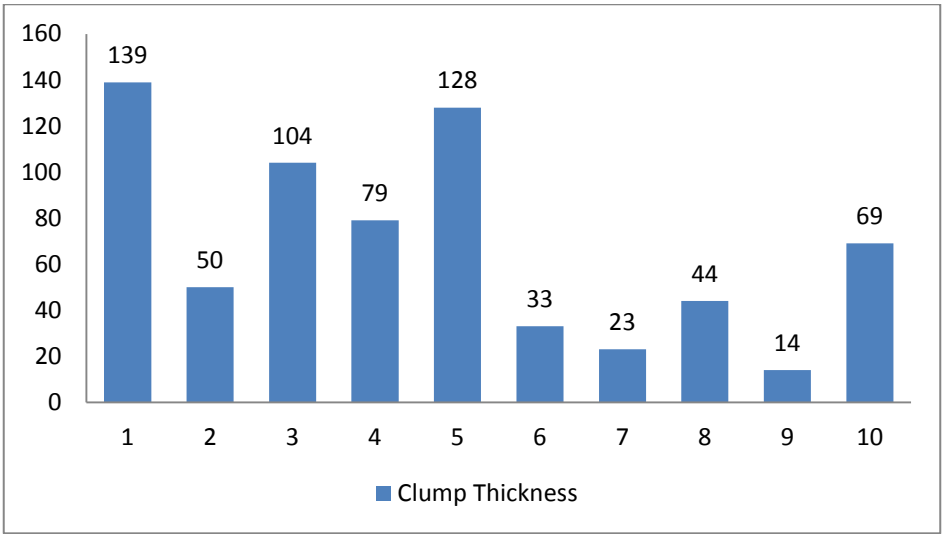



**Figure 1.** Histogram for the clump thickness attributes in the training data. The average is 4.442167.

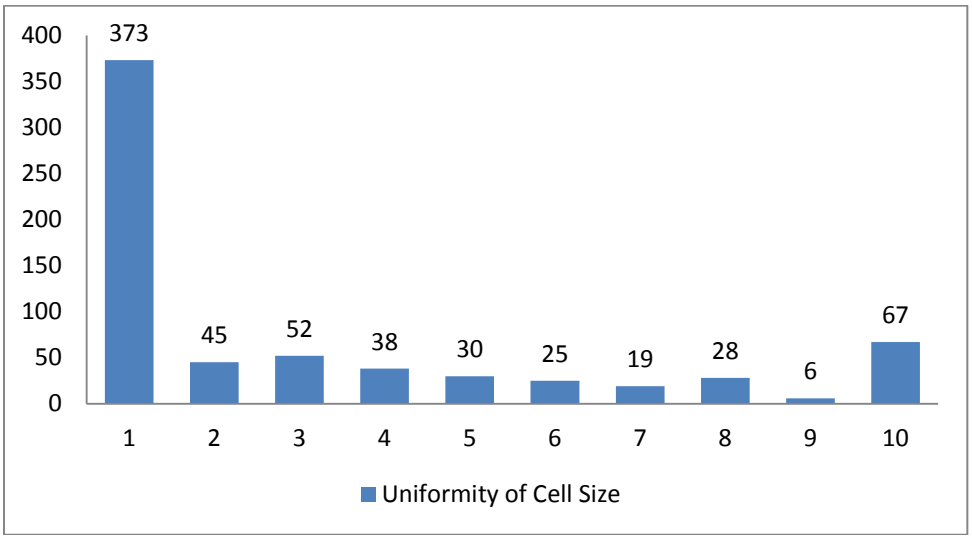



**Figure 2.** Histogram for the uniformity of Cell Size thickness attributes in the training data. The average is 3.150805.

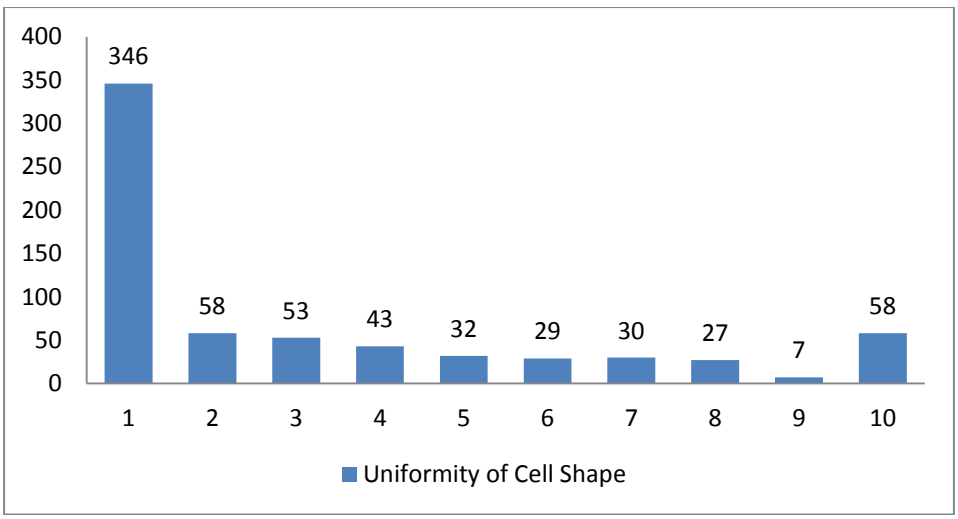



**Figure 3.** Histogram for the uniformity of Cell Shape thickness attributes in the training data. The average is 3.215227.

.

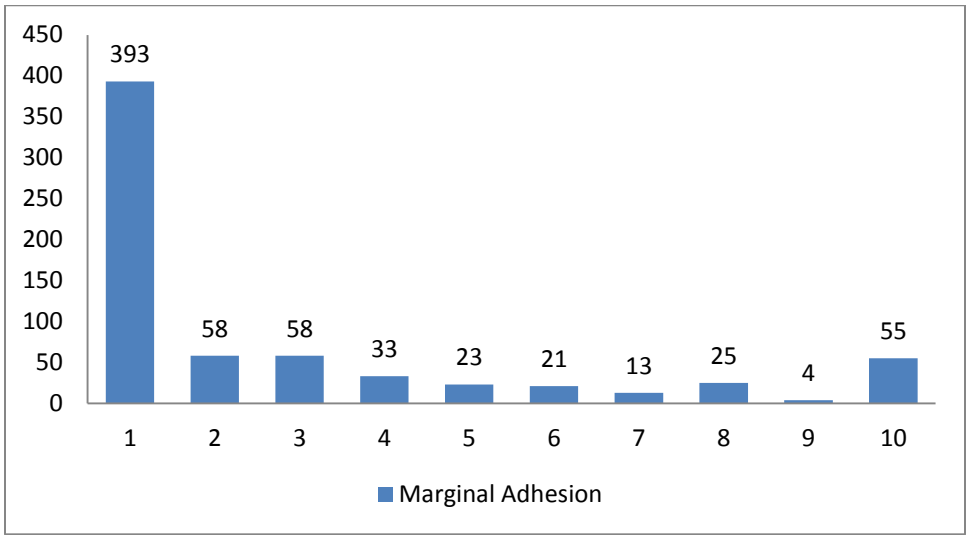



**Figure 4.** Histogram for the marginal adhesion attributes in the training data. The average is 2.830161.

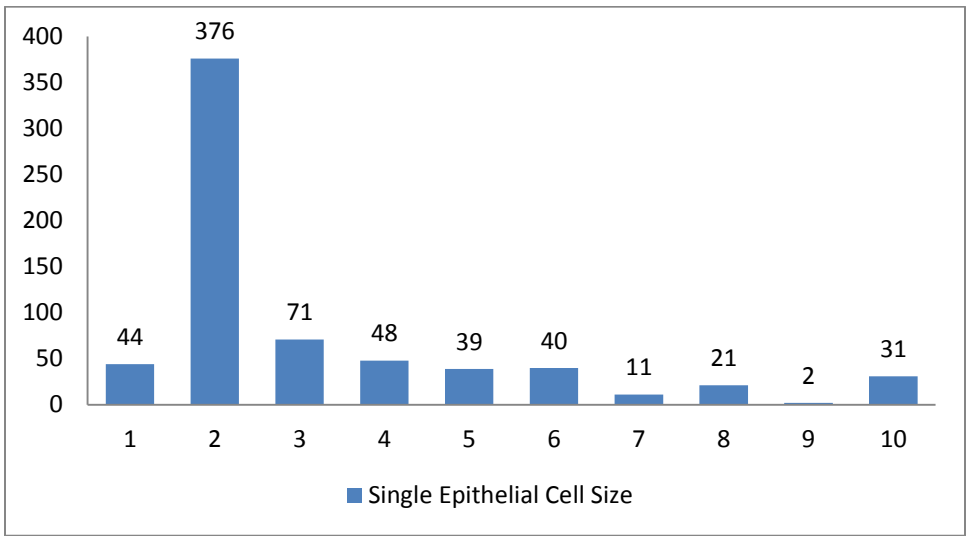



**Figure 5.** Histogram for the single epithelial cell size attributes in the training data. The average is 3.234261.

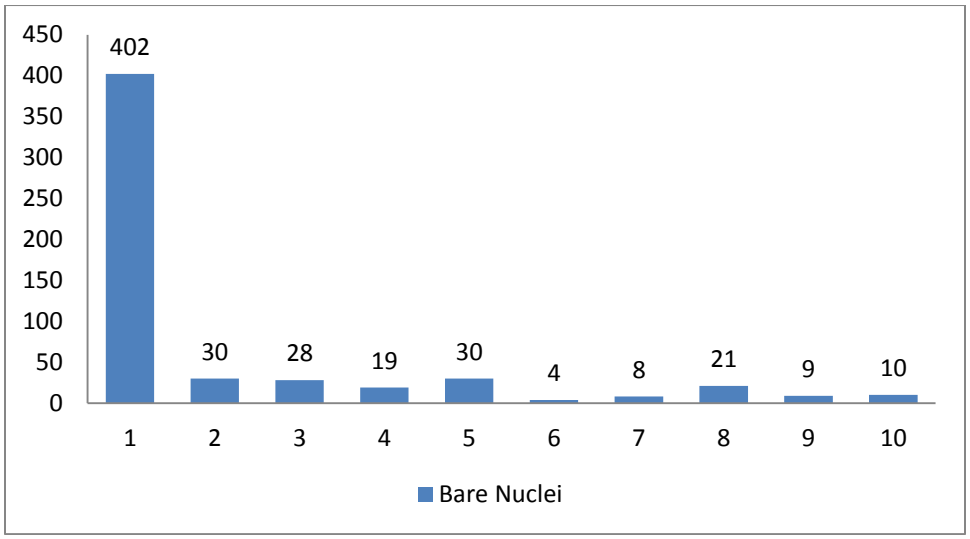



**Figure 6.** Histogram for the bare nuclei in the training data. The average is 2.14082.

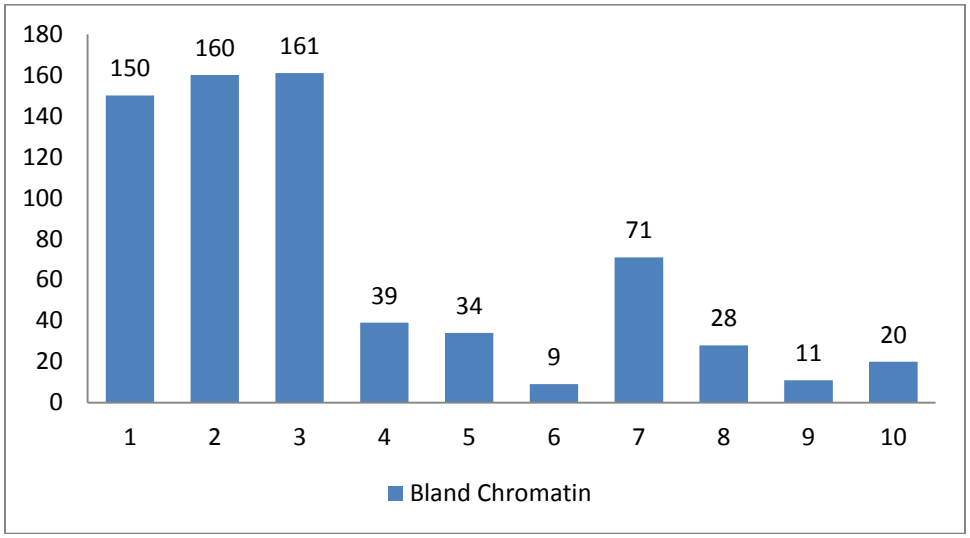



**Figure 7.** Histogram for the bland chromatin attributes in the training data. The average is 3.445095.

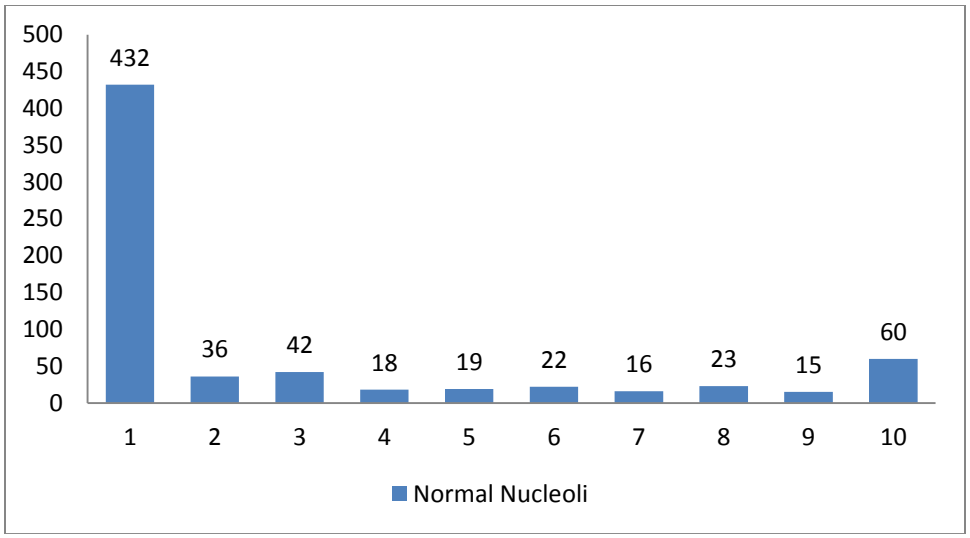



**Figure 8.** Histogram for the normal nucleoli attributes in the training data. The average is 2.869693.

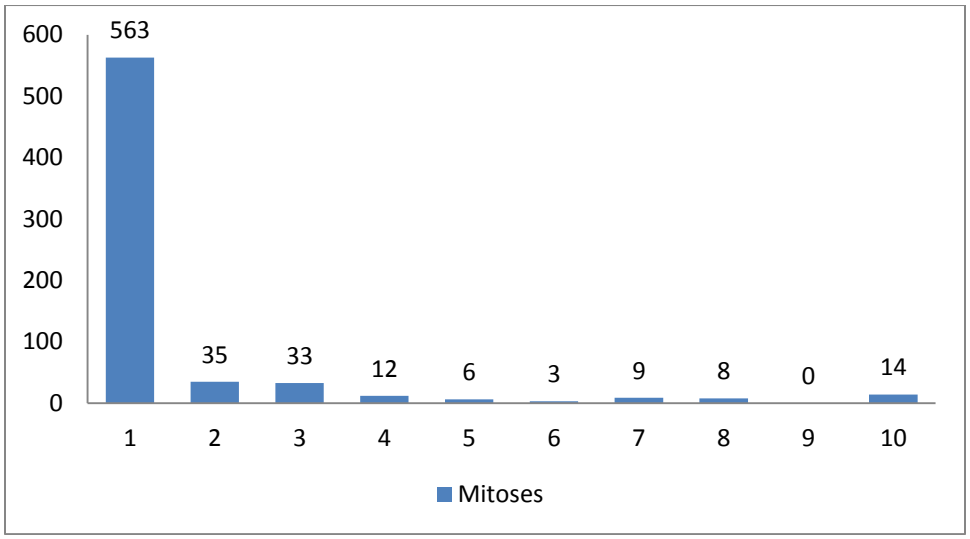



**Figure 9.** Histogram for the mitoses attributes in the training data. The average is 1.6032211.

```
>> net.IW{1,1}

ans =

    0.1261    0.1171    0.1283    0.0956    0.0751   19.0044    0.1016    0.1435    0.0820
   19.0350   19.0348    0.0000   -0.0002   -0.0001    0.0002   -0.0000    0.0000   -0.0001
   19.0349    0.0000   19.0348   -0.0001   -0.0001    0.0002   -0.0000    0.0000   -0.0001
   19.0348    0.0000    0.0000   19.0349    0.0000   -0.0000   -0.0000    0.0000   -0.0000
   19.0350    0.0006    0.0004    0.0007   19.0363   -0.0008   -0.0002   -0.0010   -0.0006
   19.0348    0.0000    0.0000   -0.0000   -0.0000    0.0000   19.0348    0.0000   -0.0000
   19.0348    0.0000    0.0000   -0.0000   -0.0000    0.0000   -0.0000   19.0348   -0.0000
   19.0348    0.0000    0.0000   -0.0000   -0.0000    0.0000   -0.0000    0.0000   19.0348
   -0.0011   19.0354   -0.0001    0.0000    0.0008    0.0013   19.0359    0.0006   -0.0008
    0.0003   19.0359    0.0011   -0.0007    0.0003    0.0027   -0.0003   19.0360   -0.0010
    0.0000   19.0348    0.0000   -0.0000   -0.0000   -0.0000   -0.0000    0.0000   19.0348
   -0.0008   -0.0001   19.0355   -0.0008   -0.0002    0.0026   19.0362   -0.0017   -0.0010
    0.0003    0.0009   19.0356   -0.0006    0.0002    0.0022   -0.0003   19.0357   -0.0008
   -0.0000   -0.0000   19.0348    0.0000   -0.0000   -0.0000    0.0000   -0.0000   19.0348
   -0.0250    0.0096    0.0091   19.1120    0.0202   -0.0412   -0.0099   19.0361   -0.0290
    0.0000    0.0000    0.0000    0.0000   19.0348   -0.0000   -0.0000   -0.0000   19.0348
  -19.0348   -0.0000  -19.0348   -0.0000   -0.0000    0.0000   -0.0000   -0.0000   -0.0000
  -19.0350   -0.0003   -0.0003  -19.0352   -0.0002    0.0001   -0.0002   -0.0004   -0.0002
  -19.0347    0.0001    0.0001    0.0001  -19.0347   -0.0001    0.0001    0.0001    0.0001
  -19.0349    0.0000    0.0000    0.0002    0.0001  -19.0349   -0.0000    0.0000   -0.0001
  -19.0347    0.0001    0.0001    0.0001    0.0001   -0.0001  -19.0347    0.0001    0.0001
  -18.9510   -0.0999   -0.0388   -0.0580   -0.0338   -0.2651   -0.1139  -18.8667    0.0280
  -19.0348    0.0002    0.0001    0.0001    0.0002    0.0000    0.0002    0.0002  -19.0348
```

**Figure 10.** The input-hidden weights

```
>> net.LW{2,1}

ans =

  Columns 1 through 9

   -0.0607    0.0607    0.0607    0.0607    0.0607    0.0607    0.0607    0.0607    0.0274
   18.8940   19.2835   18.9915   19.2846   19.2846   19.2839   19.2836   19.2836   19.2836

  Columns 10 through 18

    0.0607    0.0274    0.0604    0.0607    0.0607    0.0607    0.0607   19.0955   19.0955
   19.2836   19.2836   19.2857   19.2835   19.2836   19.4734   19.2836   -0.3980   -0.4754

  Columns 19 through 23

   19.0955   19.0955   19.0955   19.0102   19.0955
    0.0762    0.2555   -0.0332   -0.2685    0.0414
```

**Figure 11.** The hidden-output weights

```
>> net.b{1}'

ans =

  Columns 1 through 9

   -0.1761  -18.0013  -18.2638  -17.9611  -17.9594  -17.9882  -17.9997  -17.9991  -17.9994

  Columns 10 through 18

  -18.0009  -17.9991  -17.9183  -18.0016  -17.9991  -17.7604  -17.9991  -17.9908  -17.9596

  Columns 19 through 23

  -17.9965  -17.9968  -18.0051  -18.9327  -17.9995
```

**Figure 12.** The thresholds of neurons in the hidden layer

```
>> net.b{2}'

ans =

  107.9341  269.7381
```

**Figure 13.** The thresholds of neurons in the output layer