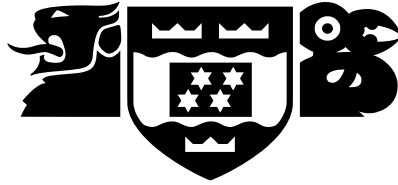


VICTORIA UNIVERSITY OF WELLINGTON  
*Te Whare Wānanga o te Ūpoko o te Ika a Māui*



School of Engineering and Computer Science  
*Te Kura Mātai Pūkaha, Pūrorohiko*

PO Box 600  
Wellington  
New Zealand

Tel: +64 4 463 5341  
Fax: +64 4 463 5045  
Internet: [office@ecs.vuw.ac.nz](mailto:office@ecs.vuw.ac.nz)

## **Logical Neural Networks: Opening the black box**

Daniel Thomas Braithwaite

Supervisor: Marcus Frean

Submitted in partial fulfilment of the requirements for  
Bachelor of Science with Honours in Computer Science.

### **Abstract**

Artificial Neural Networks (ANN's) are powerful because of their ability to approximate any function and generalize to unseen data, however they are essentially a black box, any knowledge learnt by an ANN is difficult to interpret. The ANN's presented in this report are more interpretable than standard Multilayer Perceptron Networks (MLPN's), moreover they have statistically equivalent performance and generalization to MLPN's when trained over boolean truth tables.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>3</b>
2.1	Core Concepts . . . . .	3
2.1.1	CNF & DNF . . . . .	3
2.1.2	CNF & DNF from Truth Table . . . . .	3
2.2	Literature Review . . . . .	4
<b>3</b>	<b>Foundation of Logical Normal Form Networks</b>	<b>8</b>
3.1	Noisy Gate Parametrisation . . . . .	10
3.2	Training LNF Networks . . . . .	11
3.3	LNF Network Performance . . . . .	11
3.4	LNF Network Generalization . . . . .	13
3.5	LNF Network Rule Extraction . . . . .	13
<b>4</b>	<b>Investigation of Logical Normal Form Networks</b>	<b>15</b>
4.1	Developing Applications . . . . .	15
4.1.1	Multi-class Classification . . . . .	15
4.1.2	Features with Continuous Domains . . . . .	17
	<b>Appendices</b>	<b>19</b>
<b>A</b>	<b>Tables</b>	<b>20</b>

# Chapter 1

## Introduction

Artificial Neural Networks (ANN's) are commonly used to model supervised learning problems. NN's often achieve higher accuracy than other methods because they are able to approximate any continuous function. A well trained NN can generalize well but it is very difficult to interpret how the network is operating, this is called the black-box problem.

There are a number of motivations for wanting to alleviate the black-box problem. If an ANN is able to provide an explanation for its output a deeper understanding of the problem can be developed, the rules or patterns learnt by an NN could represent some knowledge about the data which has not yet been identified. Another possibility is that the NN is being implemented to operate a systems which involves the safety of humans, a situation which is becoming more common place, in this context being able to inspect the ANN is a necessary part of ensuring the system is safe, an ANN could potentially make dangerous decisions in situations where the action taken must be extrapolated [1].

Consider some function  $f$ , then the operation of restricting a neuron to  $f$  is given by only allowing a function given by  $f$  operating on the set of weighted inputs (each input multiplied by a corresponding weight) to be learnt. By restricting the function set that each neuron can learn is it possible to create a more interpretable network? This report develops a class of networks where the function space of each neuron is restricted to be a predefined operation which can be interpreted as Boolean expression i.e. an AND or OR of its inputs.

Boolean functions are by nature discrete, as such do not have a continuous differential, making them unsuitable for training with an algorithm such as Backpropagation. This report makes use of Noisy-OR and Noisy-AND neurons [4], which are generalized continuous parametrisations of OR and AND gates.

ANN's consisting of Noisy neurons are called Logical Neural Networks (LNN's). LNN's have been used to classify the MINST dataset with promising results, while not being able to achieve a state of the art accuracy they have been shown to yield simpler (sparser) and more interpretable weight vectors [4]. It was also shown that using Noisy neurons in combination with standard sigmoid units results in poor performing networks without the benefit of being more interpretable [4], for this reason networks of this kind are not considered.

This report takes a different approach to using these interpretable Noisy neurons, instead they are placed in specific configurations which allow learning Conjunctive or Disjunctive Normal Form expressions, these are called Logical Normal Form Networks (LNFN's) and are a subset of LNN's.

This report demonstrates by experimentation that when provided a complete truth table for a boolean expression, the performance of LNFN's has no statistically significant differences to that of a Multi-Layer Perceptron (MLPN) Network. The LNFN's are also able to generalize, obtaining statistically equivalent performance to a MLPN when given incomplete truth tables.

By inspecting the weights of each Noisy neuron it is possible to determine a relationship between the inputs and output. Rule extraction algorithms exist as a method to extract knowledge from NN's [1], if LNFNs are more interpretable then are rules able to be extracted from them? When first considering problems with Boolean inputs and outputs, if a low enough error is achieved when training an LNFN over a complete truth table, it is possible to extract boolean rules from each neuron, consequently it is possible to obtain a boolean expression which represents the ANN and original truth table. Training over entire a complete data set is an unlikely scenario, this report investigates what effect training with incomplete truth tables has on any extracted formula. Only being able to apply LNFNs to Boolean problems does not make them very useful, this report investigates ways these networks can be applied to problems with inputs in a continuous domain.

The restriction placed on the function space of each neuron, while improving the interpretability, intuitively will also hinder their ability to be universal approximators. Along with the investigation of LNFNs and their potential, the limitations are also explored.

## Chapter 2

# Background

### 2.1 Core Concepts

#### 2.1.1 CNF & DNF

A boolean formula is in Conjunctive Normal Form (CNF) if and only if it is a conjunction (and) of clauses. A clause in a CNF formula is given by a disjunction (or) of literals. A literal is either an atom or the negation of an atom, an atom is one of the variables in the formula.

Consider the boolean formula  $\neg a \vee (b \wedge c)$ , the CNF is  $(\neg a \vee b) \wedge (\neg a \vee c)$ . In this CNF formula the clauses are  $(\neg a \vee b)$ ,  $(\neg a \vee c)$ , the literals used are  $\neg a$ ,  $b$ ,  $c$  and the atoms are  $a$ ,  $b$ ,  $c$ .

A boolean formula is in Disjunctive Normal Form (DNF) if and only if it is a disjunction (or) of clauses. A DNF clause is a conjunction (and) of literals. Literals and atoms are defined the same as in CNF formulas.

Consider the boolean formula  $\neg a \wedge (b \vee c)$ , the DNF is  $(\neg a \wedge b) \vee (\neg a \wedge c)$ .

#### 2.1.2 CNF & DNF from Truth Table

Given a truth table representing a boolean formula, constructing a DNF formula involves taking all rows which correspond to True and combining them with an OR operation. To construct a CNF one combines the negation of any row which corresponds to False by an OR operation and negates it.

**Theorem 2.1.1.** The maximum number of clauses in a CNF or DNF formula is  $2^n$

*Proof.* Assume the goal is to find the CNF and DNF for a Boolean formula B of size  $n$ , for which the complete truth table is given. The truth table has exactly  $2^n$  rows.

First assume a CNF is being constructed, this is achieved by taking the OR of the negation of all rows corresponding to False, the NOT operation leaves the number of clauses unchanged. At most there can be  $2^n$  rows corresponding to False, consequently there are at most  $2^n$  clauses in the CNF.

A similar argument shows that the same holds for DNF. □

## 2.2 Literature Review

A survey in 1995 focuses on rule extraction algorithms [1], identifying the reasons for needing these algorithms along with introducing ways to categorise and compare them. Motivation behind scientific study is always crucial so why is understanding the knowledge contained inside Artificial Neural Networks's (ANN's) important? The key points identified are that the ANN might of discovered some rule or patten in the data which is currently not known, being able to extract these rules would give humans a greater understanding of the problem. Another, perhaps more significant reason is the application of ANN's to systems which can effect the safety of human lives, i.e. Aeroplanes, Cars. If using an ANN in the context of a system involving human safety it is important to be certain of the knowledge inside the ANN, to ensure that the ANN wont take any dangerous actions.

There are three categories that rule extraction algorithms fall into [1]. An algorithm in the **decompositional** category focuses on extracting rules from each hidden/output unit. If an algorithm is in the **pedagogical** category then rule extraction is thought of as a learning process, the ANN is treated as a black box and the algorithm learns a relationship between the input and output vectors. The third category, **electic**, is a combination of decompositional and pedagogical. Electic accounts for algorithms which inspect the hidden/output neurons individually but extracts rules which represent the ANN globally [8].

To further divide the categories two more classifications are introduced. One measures the portability of rule extraction techniques, i.e. how easily can they be applied to different types of ANN's. The second is criteria to assess the quality of the extracted rules, these are accuracy, fidelity, consistency, comprehensibility [1].

1. A rule set is **Accurate** if it can generalize, i.e. classify previously unseen examples.
2. The behaviour of a rule set with a high **fedelity** is close to that of the ANN it was extracted from.
3. A rule set is **consistent** if when trained under different conditions it generates rules which assign the same classifications to unseen examples.
4. The measure of **comprehensibility** is defined by the number of rules in the set and the number of literals per rule.

In 1996 a class of networks, called Logical Normal Form Networks (LNFNs), where developed [2], focusing on learning the underlying CNF or DNF for a boolean expression which describes the problem. The approach relies on a specific network configuration along with restriction the function space of each neuron, allowing them to only perform an OR or AND on a subset of their inputs, such OR and AND neurons are called Disjunctive and Conjunctive retrospectively. If the trained network is able to achieve a low enough accuracy then rules can be extracted from the network in terms of a Boolean CNF or DNF expression [2].

The algorithm which extracts rules from LNFNs would be Electic and certainly is not Portable as the algorithm is specific to the LNFN architecture. It is not possible to further classify the rule extraction algorithm as the research developing it lacks any experimental results, much justification is also missing making the LNFNs difficult to reproduce.

In 2016 the concept of Noisy-OR and Noisy-AND neurons were described [4], networks containing such Noisy neurons are called Logical Neural Networks (LNNs). Pure LNNs contain only Noisy neurons. This report aims to re-derive LNNs as a subset of Pure LNNs, that is using only Noisy neurons.

Noisy neurons are derived from the Noisy-OR relation [4], developed by Judea Pearl [7], a concept in Bayesian Networks. A Bayesian Network represents the conditional dependencies between random variables in the form of a directed acyclic graph.

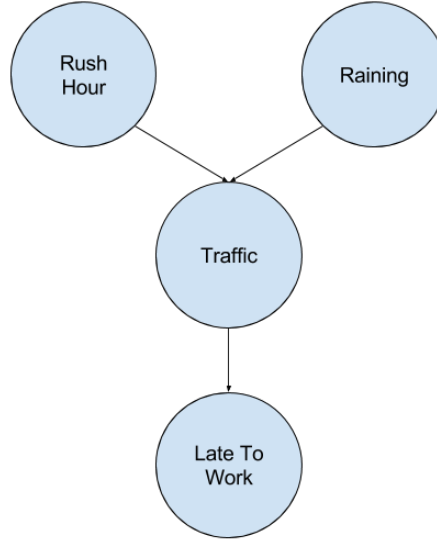


Figure 2.1

Figure 2.1 is a Bayesian network, it demonstrates the dependency between random variables "Rush Hour", "Raining", "Traffic", "Late To Work". The connections show dependencies i.e. Traffic influences whether you are late to work, and it being rush hour or raining influences whether there is traffic.

Consider a Bayesian Network having the following configuration, take some node  $D$  with  $S_1, \dots, S_n$  as parents i.e.  $S_i$  influences the node  $D$ , each  $S_i$  is independent from all others. The relationship between  $D$  and its parents is if  $S_1$  OR ... OR  $S_n$  is true then  $D$  is true. Let  $\epsilon_i$  be the uncertainty that  $S_i$  influence  $D$  then  $P(D = 1 | S_1 = 1, \dots, S_n = 1)$  can be defined.

$$P(D = 1 | S_1 = 1, \dots, S_n = 1) = 1 - \prod_{i=1}^n \epsilon_i \quad (2.1)$$

Equation 2.1 shows the noisy or relation [4]. In the context of a neuron, the inputs  $x_1, \dots, x_n$  represent the probability that inputs  $1, \dots, n$  are true. Consider the output of a neuron as conditionally dependent on the inputs, in terms of a Bayesian Network each  $x_i$  is a parent of the neuron. Each  $\epsilon_i$  is the uncertainty as to whether  $x_i$  influences the output of the neuron. How can weights and inputs be combined to create a final activation value for the neuron. First consider a function  $f(\epsilon, x)$  which computes the irrelevance of input  $x$ . Some conditions that can be placed on  $f$  are given in [4]. (1)  $\epsilon = 1$  means that  $f(\epsilon, x) = 1$ , (2)  $x = 1$  means that  $f(\epsilon, x) = 1$ , (3) Monotonically increasing in  $\epsilon$  and decreasing in  $x$ . Let  $f(x, \epsilon) = \epsilon^x$ . The definitions for Noisy-OR and Noisy-AND gates can now be given.

**Definition 2.2.1.** A **Noisy-OR** Neuron has weights  $\epsilon_1, \dots, \epsilon_n \in (0, 1]$  which represent the irrelevance of corresponding inputs  $x_1, \dots, x_n \in [0, 1]$ . The activation of a Noisy-OR Neurons is.

$$a = 1 - \prod_{i=1}^p (\epsilon_i^{x_i}) \cdot \epsilon_b \quad (2.2)$$

**Definition 2.2.2.** A **Noisy-AND** Neuron has weights  $\epsilon_1, \dots, \epsilon_n \in (0, 1]$  which represent the irrelevance of corresponding inputs  $x_1, \dots, x_n \in [0, 1]$ . The activation of a Noisy-AND Neurons is.

$$a = \prod_{i=1}^p (\epsilon_i^{1-x_i}) \cdot \epsilon_b \quad (2.3)$$

Both these parametrisations reduce to discrete logic gates when there is no noise, i.e.  $\epsilon_i = 0$  for all  $i$ .

ANN's containing of Noisy-OR and Noisy-AND neurons are called Logical Neural Networks (LNN's), if the network consists of only Noisy neurons then it a pure LNN. LNN's have been applied to the MINST dataset with promising results. Experiments with different combinations of logical and standard (sigmoid, soft-max) neurons have shown that pure LNN's where able to achieve an error of 8.67%, where a standard perceptron/softmax network was able to achieve an error of 3.13%. This reduction in performance does not come without reward, the pure LNN yields a simpler (sparser) and more interpretable network [4]. Training LNN's which are not pure have been shown to have reduced performance (compared to standard ANN's) and no interpretability benefit.

While the concept given in [2] is the foundation for the work given in this report, the approach presented is different that what has been done. A simpler parametrisation for disjunctive and conjunctive neurons is used, along with better justification and more investigation of the capabilities of LNFN's.

LNFNs are based off the idea of learning an underlying boolean formula, what happens when learning problems which consist of continous features? One option is to simply train the networks using the continuous features, the other is to descretise any continuous value.

There are a number of categories used to sort various descretization algorithms [6], three of these categories are defined below.

1. Supervised vs. Unsupervised: Supervised descretization algorithms use class information, as opposed to unsupervised which do not.
2. Dynamic vs. Static: Dynamic methods descretize features as the classifier is being built. A static algorithm performs all descretization before the lerning begins.
3. Global vs. Local: Global methods descretize the entire instance space where as local only descretize a subset of the total space.

The types of descretization algorithms which will be considered are Static and Global. Dynamic does not make a lot of sense when the data is used in training an ANN and a Local does not make sense if we are using a Static method. As for Supervised vs. Unsupervised



it does not matter, though since we do have access to the labels using them could provide better performance.

Basic discretizing methods are Equal width and frequency binning, these are unsupervised algorithms. Equal width divides the interval up into  $K$  equally sized bins, where as equal frequency divides the interval into  $K$  bins where each one has the same number of instances. While implementation is easy it is not without a cost, namely results could be poor if the distribution of features is not uniform [6]. The two methods also require that  $K$  be pre defined, this could require an expensive trial and error process. [6].

A supervised technique is called Recursive Minimal Entropy Partitioning (RMEP). Let  $S = \{s_1, \dots, s_n\}$  be a set of instances, each with  $n$  attributes. The goal of RMEP is to partition the range of each attribute. The entropy of a set is given by equation 2.4

$$E(S) = \sum_i p_i \cdot \log(p_i) \quad (2.4)$$

## Chapter 3

# Foundation of Logical Normal Form Networks

Consider problems with a boolean expression describing the relation ship between binary inputs and outputs. This is certainly a restricted space of problems, however its a logical place to start if defining the activation functions of neurons as a "boolean like" function.

With this restriction in place any problem must be described by a boolean expression, this information alone does not help with constructing interpretable networks. It is known that any boolean expression has a CNF and DNF. It is possible to construct networks which can learn the underlying CNF or DNF, such networks are called Logical Normal Form Networks (LNFNs) [2]. The research developing LNFNs has little justification for key decisions, consequently is difficult to understand and reproduce.

Using the idea of LNFNs [2] and Pure Logical Neural Networks [4] new definitions are given for LNFNs in terms of Noisy-OR and Noisy-AND neurons.

**Definition 3.0.1.** A **CNF-Network** is a three layer network where neurons in the hidden layer consist solely of Noisy-OR's and the output layer is a single Noisy-AND.

**Definition 3.0.2.** A **DNF-Network** is a three layer network where neurons in the hidden layer consist solely of Noisy-AND's and the output layer is a single Noisy-OR.

**Definition 3.0.3.** A **LNF-Network** is a DNF or CNF Network

A CNF or DNF formula contains clauses of literals which is either an atom or a negation of an atom. To account for this the number of inputs to the network will be doubled, the inputs will be all the atoms and negations of the atoms, i.e. if  $x_1, x_2$  are the atoms then  $x_1, \neg x_1, x_2, \neg x_2$  are the inputs to the network.

It must also be determined how many hidden units the LNFN will have, it is known that  $2^n$ ,  $n$  being the number of atoms, is an upper bound on the number of clauses needed in a CNF and DNF formula (see Theorem 2.1.1).

**Theorem 3.0.1.** Let  $T$  be the complete truth table for the boolean formula  $B$ . Let  $L$  be an LNFN, if  $L$  has  $2^n$  hidden units then there always exists a set of weights for  $L$  which correctly classifies any assignment of truth values to atoms.

*Proof.* Let  $T$  be the truth table for a boolean function  $B$ . The atoms of  $B$  are  $x_1, \dots, x_n$ .  $T$  has exactly  $2^n$  rows. Construct an LNFN,  $L$ , in the following manner.  $L$  has  $2^n$  hidden units and

by definition L has one output unit. The inputs to L are  $i_1, \dots, i_{2n}$  where  $i_1, i_2$  represent  $x_1, \neg x_1$  and so on. Let  $\epsilon_b = 1$  for every neuron.

Let  $h_k$  denote hidden unit k.  $h_k$  has the weights  $\epsilon_{k,1}, \dots, \epsilon_{k,2n}$ , where  $\epsilon_{k,m}$  represents input  $i_m$ 's relevance to the output of  $h_k$ . Similarly the output unit  $o$  has weights  $\mu_1, \dots, \mu_{2^n}$  where  $\mu_m$  represents the relevance of  $h_m$  to the output of  $o$ .

Assume L is a DNF Network. Starting from row one of the table T, to row  $2^n$ . If row  $a$  corresponds to False then set  $\mu_a = 1$  (i.e. hidden node  $a$  is irrelevant), otherwise the row corresponds to True, then  $\mu_a = Z$ , where  $Z$  is a value close to 0 (any weight for a Noisy neuron can't be exactly 0). For each  $\epsilon_{a,m}$  if the corresponding literal occurs in row  $a$  of the truth table then  $\epsilon_{a,m} = Z$  other wise  $\epsilon_{a,m} = 1$ .

**Claim:** For some assignment to the atoms of B,  $x_1 = v_1, \dots, x_n = v_n$  where  $v_i \in \{0, 1\}$ . Then  $L(i_1, \dots, i_{2n}) = B(x_1, \dots, x_n)$ .

Assume  $B(x_1, \dots, x_n) = 1$  for the assignment  $x_1 = v_1, \dots, x_n = v_n$  corresponding to row  $a$  of T. Then if  $i_k$  is not considered in row  $a$  then  $\epsilon_{a,k} = 1$  and if it is present then  $i_k = 1$ . The output of  $h_a$  is given by

$$\begin{aligned} &= \prod \epsilon_{a,m}^{1-i_m} \\ &= Z^{\sum_{i_k=1} (1-i_k)} \\ &= Z^0 \end{aligned}$$

Demonstrating that  $\lim_{Z \rightarrow 0} \text{Out}(h_a) = \lim_{Z \rightarrow 0} Z^0 = 1$ . Consider the activation of  $o$ , it is known that  $\mu_a = Z$  consequently  $\lim_{Z \rightarrow 0} \mu_a^{h_a} = \lim_{Z \rightarrow 0} Z^1 = 0$ , therefore

$$\lim_{Z \rightarrow 0} \text{Out}(o) = 1 - \prod_{m=1}^{2^n} \mu_m^{h_m} \quad (3.1)$$

$$= 1 - 0 = 1 \quad (3.2)$$

Therefore  $L(i_1, \dots, i_{2n}) = 1$ . Alternatively if  $B(x_1, \dots, x_n) = 0$  then no hidden neuron will have activation 1, this can be demonstrated by considering that any relevant neuron (i.e. corresponding  $\mu \neq 1$ ) will have some input weight pair of  $i_m \epsilon_m$  such that  $\epsilon_m^{i_m} = 0$ . Consequently it can be said that for all  $m$   $\mu_m^{h_m} = \mu_m^0 = 1$ , therefore the output unit will give 0, as required.

Now assume that L is a CNF Network. The weights can be assigned in the same manner as before, except rather than considering the rows that correspond to True the negation of the rows corresponding to False are used. If a row  $a$  corresponds to True then  $\mu_a = 1$ , otherwise  $\mu_a = Z$  and for any literal present in the row then the input to L which corresponds to the negated literal has weight  $Z$ , all other weights are 1.

**Claim:** For some assignment to the atoms of B,  $x_1 = v_1, \dots, x_n = v_n$  where  $v_i \in \{0, 1\}$ . Then  $L(i_1, \dots, i_{2n}) = B(x_1, \dots, x_n)$ .

In this configuration it must be shown that every hidden neuron fires when the network is presented with a variable assignment which corresponds to True and there is always at least one neuron which does not fire when the assignment corresponds to False. Assume for

a contradiction that for a given assignment  $B(x_1, \dots, x_n) = 1$  but  $L(i_1, \dots, i_{2n}) = 0$ . Then there is at least one hidden neuron which does not fire. Let  $h_a$  be such a neuron. Consequently for any input weight combination which is relevant  $\epsilon_{a,m}^{i_m} = 1$ , so  $i_m = 0$  for any relevant input. Let  $i_{r_1}, \dots, i_{r_k}$  be the relevant inputs then  $i_{r_1} \vee \dots \vee i_{r_k} = \text{False}$ , so  $\neg(\neg i_{r_1} \wedge \dots \wedge \neg i_{r_k}) = \text{False}$ , a contradiction as then  $B(x_1, \dots, x_n)$  would be False.

Now assume for a contradiction  $B(x_1, \dots, x_n) = 0$  but  $L(i_1, \dots, i_{2n}) = 1$ . Then there exists some  $h_a$  with output 1 where it should be 0. Consequently there exists at least one input/weight pair with  $\epsilon_{a,m}^{i_m} = 1$  that should be 0. Let  $i_{r_1}, \dots, i_{r_k}$  be all the relevant inputs, at least one relevant input is present  $i_r$ . Consequently  $i_{r_1} \vee \dots \vee i_{r_k} = \text{True}$ , therefore  $\neg(\neg i_{r_1} \wedge \dots \wedge \neg i_{r_k}) = \text{True}$ , a contradiction as then  $B(x_1, \dots, x_n)$  is True.  $\square$

Theorem 3.0.1 provides justification for using  $2^n$  hidden units, it guarantees that there at least exists an assignment of weights yielding a network that can correctly classify each item in the truth table.

### 3.1 Noisy Gate Parametrisation

The parametrisation of Noisy gates require weight clipping, an expensive operation. A new parametrisation is derived that implicitly clips the weights. Consider that  $\epsilon \in (0, 1]$ , therefore let  $\epsilon_i = \sigma(w_i)$ , these  $w_i$ 's can be trained without any clipping, after training the original  $\epsilon_i$ 's can be recovered.

Now these weights must be substituted into the Noisy activation. Consider the Noisy-OR activation.

$$\begin{aligned}
a_{or}(X) &= 1 - \prod_{i=1}^p (\epsilon_i^{x_i}) \cdot \epsilon_b \\
&= 1 - \prod_{i=1}^p (\sigma(w_i)^{x_i}) \cdot \sigma(b) \\
&= 1 - \prod_{i=1}^p \left( \left( \frac{1}{1 + e^{-w_i}} \right)^{x_i} \right) \cdot \frac{1}{1 + e^{-b}} \\
&= 1 - \prod_{i=1}^p ((1 + e^{-w_i})^{-x_i}) \cdot (1 + e^{-w_i})^{-1} \\
&= 1 - e^{\sum_{i=1}^p -x_i \cdot \ln(1 + e^{-w_i}) - \ln(1 + e^{-b})} \\
&\text{Let } w'_i = \ln(1 + e^{-w_i}), \quad b' = \ln(1 + e^{-b}) \\
&= 1 - e^{-(W' \cdot X + b')}
\end{aligned}$$

From a similar derivation we get the activation for a Noisy-AND.

$$\begin{aligned}
a_{and}(X) &= \prod_p^{i=1} (\epsilon_i^{1-x_i}) \cdot \epsilon_b \\
&= \prod_p^{i=1} (\sigma(w_i)^{1-x_i}) \cdot \sigma(w_b) \\
&= e^{\sum_{i=1}^p -(1-x_i) \cdot \ln(1+e^{-w_i}) - \ln(1+e^{-b})} \\
&= e^{-(W' \cdot (1-X) + b')}
\end{aligned}$$

Concisely giving equations 3.3, 3.4

$$a_{and}(X) = e^{-(W' \cdot (1-X) + b')} \quad (3.3)$$

$$a_{or}(X) = 1 - e^{-(W' \cdot X + b')} \quad (3.4)$$

The function taking  $w_i$  to  $w'_i$  is the soft ReLU function which is performing a soft clipping on the  $w_i$ 's.

## 3.2 Training LNF Networks

Using equations 3.4 and 3.3 for the Noisy-OR, Noisy-AND activations retrospectively allows LNFNs to be trained without explicit clipping. The ADAM Optimizer is used for training firstly for the convenience of an adaptive learning rate but also because it includes the advantages of RMSProp which works well with on-line (single-example) learning [3], which LNF Networks respond well to.

Preliminary testing showed that LNFN's are able to learn good classifiers on boolean gates, i.e. NOT, AND, NOR, NAND, XOR and Implies. It is also possible to inspect the trained weights and see that the networks have learnt the correct CNF or DNF representation.

## 3.3 LNF Network Performance

How do LNFNs perform against standard perceptron networks which we know to be universal function approximators. Two different perceptron networks will be used as a benchmark

1. One will have the same configuration as the LNFNs, i.e.  $2^n$  hidden neurons.
2. The other has two hidden layers, both with  $N$  neurons.

The testing will consist of selecting 5 random boolean expressions for  $2 \leq n \leq 9$  and training each network 5 times, each with random initial conditions. Figure 3.1 shows a comparison between all 4 of the networks and figure 3.2 shows just the LNFN's.

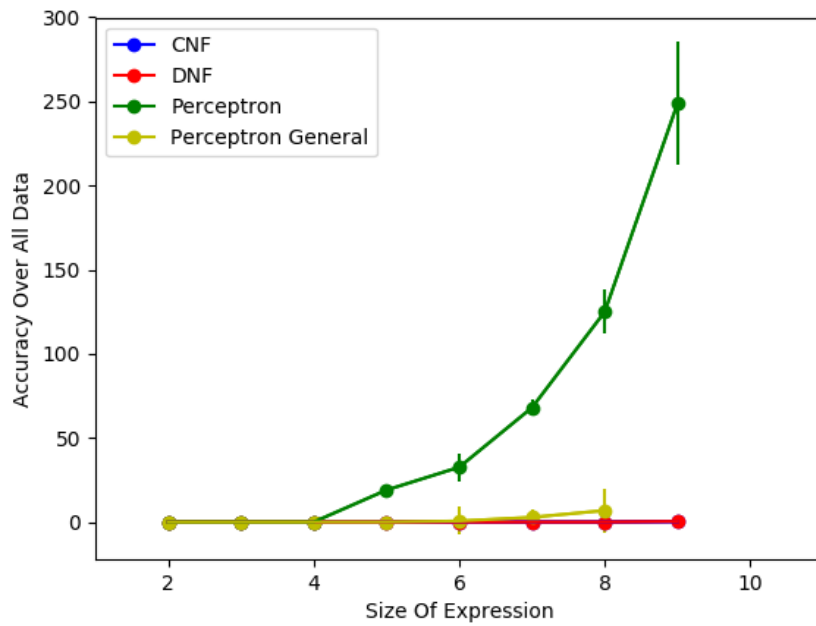


Figure 3.1

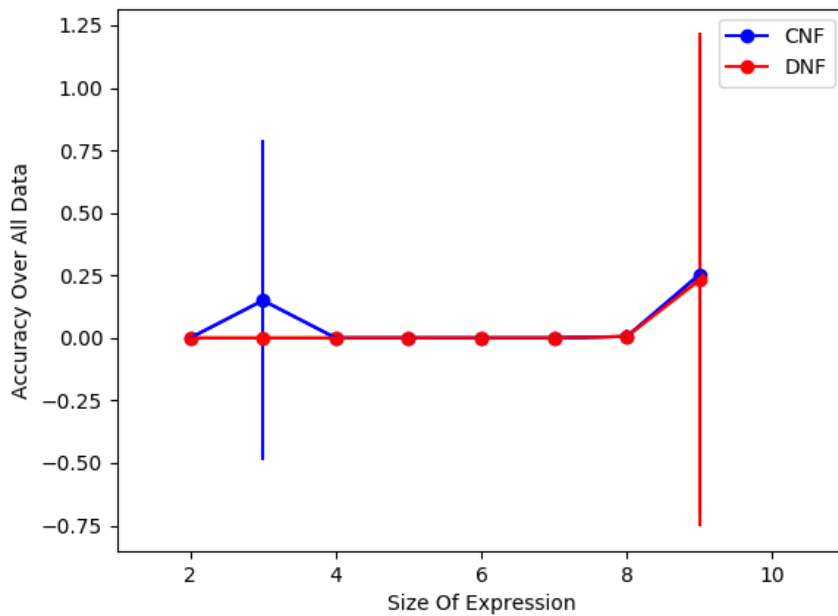


Figure 3.2

Figure 3.1 shows that neither of the perceptron networks perform as well as the LNF Networks as  $n$  increases. Figure 3.2 shows on average there are no statistically significant differences between the CNF or DNF networks. What is not present in 3.2 is that at  $n = 9$  sometimes the CNF network far out performs the DNF and visa versa, theoretically both should be able to learn any boolean expression.

### 3.4 LNF Network Generalization

These networks are able to perform as well as standard perceptron networks but so far they have been getting the complete set of data, in practice this will almost never be the case. Standard ANN's are widely used because of their ability to generalize, for LNFN's to be useful they must also be able to generalize.

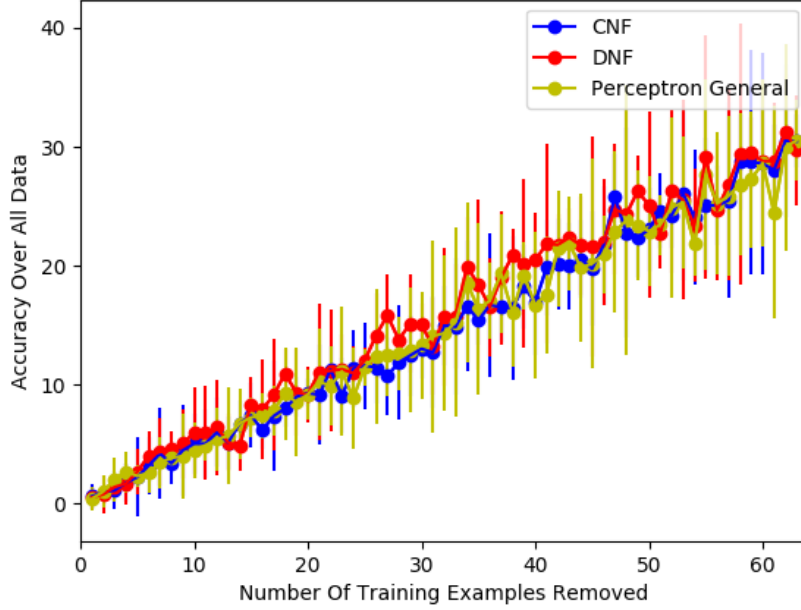


Figure 3.3

Figure 3.3 shows a comparison between the generalization ability of CNF, DNF and Perceptron networks. The graph shows the performance over all training data when successively removing elements from the training set. It demonstrates that the CNF and DNF networks generalize as well as the perceptron networks when the boolean formula has 6 inputs, this trend continues as  $n$  increases up to 9.

### 3.5 LNF Network Rule Extraction

The goal of this report is to learn rules which can be extracted from the network, consequently a method must be developed to extract the rules from LNF Network's. Take the weights of a trained LNFN, these weights can be converted back into  $\epsilon_i$ 's by apply the sigmoid function to each  $w_i$ .

As  $\epsilon_i \rightarrow 0$  then  $x_i$  becomes relevant and as  $\epsilon_i \rightarrow 1$  then  $x_i$  becomes irrelevant. If the network has learnt the correct DNF or CNF representation the for every neuron if input  $i$  is relevant then  $w_i \rightarrow -\infty$  and therefore  $\epsilon_i \rightarrow 0$ , otherwise  $x_i$  is irrelevant and  $w_i \rightarrow \infty$  meaning  $\epsilon_i \rightarrow 1$ .

Consequently in  $\epsilon$  form the weights of the LNFN are almost binary (i.e. very close to 0 or 1) and rules can be easily extracted as a boolean formula. Many clauses in the extracted

expression contain redundant terms, i.e. clauses that are a tautology or a duplicate of another, filtering these out is not an expensive operation.

When training an LNFN over the entire truth table for a boolean expression, when a low error is achieved it is possible to extract boolean formula from the network which gets can generate the original truth table. This is a necessary first step but a more important question is, can formula still be extracted from the network when the LNFN is not trained with the entire truth table? If it can then what does this formula represent?



## Chapter 4

# Investigation of Logical Normal Form Networks

### 4.1 Developing Applications

The LNFNs presented in Chapter 3 have very limited applications, the class of problems they can be applied to all have the form of binary classification where the features are Boolean. To make LNFNs more useful they need a broader scope.

#### 4.1.1 Multi-class Classification

How can LNFNs be developed] to support classification of more than two classes. If attempting to learn a Multi-class Classification problem with  $n$  distinct classes  $c_1, \dots, c_n$ , then  $n$  LNFNs would have to be trained, each a learning a binary classification problem  $c_i, \neg c_i$ .

An intuitive way to extend LNFNs to support Multi-class Classification is to add more output neurons and use One-hot encoding. If we have 3 classes then 100, 010 and 001 represent class 1, 2 and 3 retrospectively, then the LNFN would have 3 output neurons, each representing a bit in the One-hot encoded string.

**Definition 4.1.1.** The structure of an LNFN to solve an  $k$  class classification problem where each instance is described by  $n$  features has  $2n$  inputs,  $2^n$  hidden units and  $k$  output units. A final Softmax layer is added, as is convention for these types of problems.

A simple problem which lends its self naturally to this is the Lenses problem [5], a three class classification problem, each instance has 4 features, 3 of which are binary and the other has three possible values. This problem can be easily converted into one which can be used with an LNFN, each new problem instance will have 6 features, the three binary remain the same and the categorical one is expanded into 3.

How does an LNFN network perform when compared to a MLPN on the Lenses problem. The performance of the two classifiers will be compared using Leave-One-Out (LOE) Cross-Validation. The structure of the MLPN only differs in the number of hidden layers/units, there are two hidden layers, one with  $2 \cdot n$  hidden units and the other with  $n$

	Error (SSE)	Confidence Interval (95%)
CNF Net	6.663	(6.468, 7.198)
DNF Net	6.660	(6.468, 7.197)
PCEP Net	6.751	(6.468, 7.997)

Table 4.1

Table 4.1 demonstrates that the CNF & DNF Networks perform comparably to an MLPN as the confidence intervals for the error overlap.

Now that the LNFN network has three output neurons it should be possible to extract three rules describing each of the classes. Consider that each problem instance is of the following form  $\{a, b, c, d, e, f\}$  where  $a, b, c, d, e, f$  are all atoms. The following rules can be extracted from a CNFN when trained over the complete Lenses data, any duplicate clause or Tautology has been filtered out, the resultant extracted formula has also been manually simplified (so they can be displayed and understood better).

- Class 1:  $(a \vee b \vee e) \wedge (a \vee \neg d) \wedge (c \vee e) \wedge f$
- Class 2:  $(a \vee b \vee \neg c \vee d) \wedge \neg e \wedge f$
- Class 3:  $(\neg a \vee b \vee c \vee \neg f) \wedge (a \vee \neg d \vee e \vee \neg f) \wedge (\neg b \vee c \vee d \vee \neg f) \wedge (d \vee \neg e \vee \neg f)$

Immediately it is possible to find useful information about this problem that was not obvious before, namely  $\neg f = \text{True} \implies \text{Class 3}$ . Table A.1 shows these rules applied to all the problem instances in the Lenses data set, it demonstrates that these extracted rules are able to fully describe the Lenses problem.

The DNFN might be more applicable as the rules will be more insightful, given its structure as an OR of ANDs.

- Class 1:  $(a \wedge \neg b \wedge \neg c \wedge e \wedge f) \vee (\neg a \wedge \neg d \wedge e \wedge f)$
- Class 2:  $(\neg c \wedge \neg e \wedge f) \vee (c \wedge d \wedge \neg e \wedge f)$
- Class 3:  $(\neg a \wedge \neg b \wedge c \wedge \neg d) \vee (\neg a \wedge d \wedge e) \vee \neg f$

These DNF formula do not correspond to the CNF give above but this is to be expected, despite the fact that collectively all the problem instances span the space of features once they have been converted to the boolean feature form this fact is no longer true. there is a way to combine the knowledge from both the CNF and DNF formula to create resultant formula which perform better.

One possible issue that could arrive here is that the CNFN is now attempting to learn three CNF expressions with the same number of hidden neurons. The fact that meaningful rules were able to be extracted in this case could of been a coincidence, intuitively if we are learning a problem with  $k$  classes then we could need  $k \cdot 2^n$  hidden neurons which is becoming impractical.

### 4.1.2 Features with Continuous Domains

The inputs to an LNFN are allowed to be continuous but must be in the range  $[0, 1]$ , would it still be possible to extract meaningful rules from the network if the inputs are continuous? Here there are two things to investigate. What can be achieved by training an LNFN on problems with continuous features? Secondly what can be achieved by discretizing the continuous inputs and then using an LNFN to learn this new boolean problem. A simple benchmark to use is the Iris problem [5]

#### LNFNs and Continuous Features

If the features are continuous it no longer makes sense to extract rules but it could still be possible to see what inputs are considered in making a decision about the class. When training an LNFN on the Iris problem (over all problem instances) the network converges to a solution with a loss of 96.932, poor performance when compared to a perceptron network that can achieve an accuracy of 0.0.

Inspecting the class prediction of each reveals that for a problem instance that has a true class of Iris-virginica or Iris-versicolor then LNFN sometimes predicts multiple classes, this leads to the belief that these networks have issues with learning problems that are not linearly separable as the two classes which an LNFN has trouble differentiating between are not linearly separable.

The LNFN is able to learn XOR so not all problems which are not linearly separable are out of reach of LNFNs. Section 4.1.2 investigates whether it is possible to discretize the variables in such a way that makes this problem learnable by LNFNs.

#### LNFNs and Discretized Continuous Features

There are a number of methods for discretizing continuous features, there are many algorithms for performing such an operation on some data [6], too many for all to be tested. Two simple methods for discretization are Equal width or frequency binning, these methods are very naive and prone to outliers, also the number of bins must be chosen beforehand so this requires experimentation. A supervised partitioning method will also be tested, namely Recursive Minimal Entropy Partitioning.

Results from training LNFNs with discretized data results in the same issue as before, the network has problems with classifying classes that are not linearly separable.

#### Discussion of Application to Continuous Domains

The ideas explored in Sections 4.1.2 & 4.1.2 demonstrate that applying LNFNs to problems with continuous inputs is not viable.

# Bibliography

- [1] ANDREWS, R., DIEDERICH, J., AND TICKLE, A. B. Survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge-based systems* 8, 6 (1995), 373–389.
- [2] HERRMANN, C., AND THIER, A. Backpropagation for neural dnf-and cnf-networks. *Knowledge Representation in Neural Networks, S* (1996), 63–72.
- [3] KINGMA, D., AND BA, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
- [4] LAURENSEN, J. Learning logical activations in neural networks.
- [5] LICHMAN, M. UCI machine learning repository, 2013.
- [6] LIU, H., HUSSAIN, F., TAN, C. L., AND DASH, M. Discretization: An enabling technique. *Data mining and knowledge discovery* 6, 4 (2002), 393–423.
- [7] RUSSELL, S., NORVIG, P., AND INTELLIGENCE, A. A modern approach. *Artificial Intelligence. Prentice-Hall, Englewood Cliffs* 25 (1995), 27.
- [8] TICKLE, A. B., ANDREWS, R., GOLEA, M., AND DIEDERICH, J. The truth will come to light: Directions and challenges in extracting the knowledge embedded within trained artificial neural networks. *IEEE Transactions on Neural Networks* 9, 6 (1998), 1057–1068.

# Appendices

# Appendix A

## Tables

a	b	c	d	e	f	Rule Out	Expected
1	0	0	0	0	0	(0, 0, 1)	(0, 0, 1)
1	0	0	0	0	1	(0, 1, 0)	(0, 1, 0)
1	0	0	0	1	0	(0, 0, 1)	(0, 0, 1)
1	0	0	0	1	1	(1, 0, 0)	(1, 0, 0)
1	0	0	1	0	0	(0, 0, 1)	(0, 0, 1)
1	0	0	1	0	1	(0, 1, 0)	(0, 1, 0)
1	0	0	1	1	0	(0, 0, 1)	(0, 0, 1)
1	0	0	1	1	1	(1, 0, 0)	(1, 0, 0)
0	1	0	0	0	0	(0, 0, 1)	(0, 0, 1)
0	1	0	0	0	1	(0, 1, 0)	(0, 1, 0)
0	1	0	0	1	0	(0, 0, 1)	(0, 0, 1)
0	1	0	0	1	1	(1, 0, 0)	(1, 0, 0)
0	1	0	1	0	0	(0, 0, 1)	(0, 0, 1)
0	1	0	1	0	1	(0, 1, 0)	(0, 1, 0)
0	1	0	1	1	0	(0, 0, 1)	(0, 0, 1)
0	1	0	1	1	1	(0, 0, 0)	(0, 0, 1)
0	0	1	0	0	0	(0, 0, 1)	(0, 0, 1)
0	0	1	0	0	1	(0, 0, 1)	(0, 0, 1)
0	0	1	0	1	0	(0, 0, 1)	(0, 0, 1)
0	0	1	0	1	1	(1, 0, 0)	(1, 0, 0)
0	0	1	1	0	0	(0, 0, 1)	(0, 0, 1)
0	0	1	1	0	1	(0, 1, 0)	(0, 1, 0)
0	0	1	1	1	0	(0, 0, 1)	(0, 0, 1)
0	0	1	1	1	1	(0, 0, 0)	(0, 0, 1)

Table A.1: Rules extracted from CNF applied to Lenses problem