Weber State University

Master of Science in Computer Science

Thesis

Student name: Jarren Briscoe E-mail: jarren.briscoe@gmail.com

Supervisory Committee

Chair: Dr. Robert Ball E-mail: robertball@weber.edu
Member: Dr. Kyle Feuz E-mail: kylefeuz@weber.edu
Member: Dr. Brian Rague E-mail: brague@weber.edu

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

Abstract In this thesis, I discover that a neural network can be approximated as an AIG using an optimized binary decision tree (called $\mathcal{B}$) over a parsing algorithm that skips negligible neural nodes (called $\mathcal{N}$). $\mathcal{B}$ has an average complexity of $O(2^{0.5n})$ and a worst-case complexity of $O(2^n)$ while the linear algebra method would be $\Omega(2^n n \log_2 n)$ and the brute-force method takes $\Theta(2^n n)$. Creating an AIG from a single neural network picked up evolving or less important relationships. However, an intersection of several AIGs compiled from several neural networks generalizes the neural network ensemble by highlighting the important relationships.

Format This thesis is divided into five parts: introduction (part I), background (part II), methods (part III), results (part IV), and the appendix (appendix A). The introduction will give a summary of the thesis. The background will clarify definitions and mention related work. The finer details of the work I have done will be in the methods part. The results will show experiments on complexity, give a conclusion, and future work. Finally, refer to the appendix for a reminder of common abbreviations, mathematical symbols, and other definitions.

1.1 Background summary

Neural networks (NNs) are a powerful and incomprehensible machine learning technique due to their intricate structure. Reasonably, a comprehensible structure must be concise (further explained in chapter 4). The many tech-

niques to analyze neural networks taken from surveys in [4, 5] fit into three general categories: decompositional, eclectic, and pedagogical. Decompositional techniques consider individual weights, activation functions, and the finer details of the network. For example, learning each node in the neural network is a decompositional technique.

On the other hand, a pedagogical approach learns by an oracle or teacher—allowing a relatively facile change in underlying machine learning structures. For example, a simple pedagogical approach may be a decision tree whose training inputs are fed into a neural network, and the decision tree's learned class values are the output of the neural network. This simple pedagogical approach can easily transition from the neural network black box to another machine learning algorithm.

Finally, eclectic approaches consider the neural networks as a gestalt—a figure whose sum (big picture) is more valuable than the individual parts (details). As a gestalt, the details are still worth considering for this intertwined approach. An example of an eclectic approach is directing a learning algorithm over neural nodes while omitting inputs negligible to the final product. In this paper, I take an eclectic approach with localized learning of each neuron (a decompositional technique) with directions given by the latter layers (partly pedagogical).

1.2 End product

The neural networks are translated to the more comprehensible AIGs.

AIGs AIGs (And-Inverter Graphs) are formally verifiable (provable with formal methods) and legacy white box structures. The AIG is a directed and acyclic graph that consists of an input layer, a finite or null amount of intermediate layers, and an output layer. Besides the input nodes, every node in an AIG is binary and performs logical conjunction upon its two in-edges. Additionally, every edge in an AIG can invert inputs (perform a logical NOT), hence the term "And-Inverter Graph".

Software Open-source implementations of AIGs with active maintenance already exist. In my thesis, I used Mockturtle from the EPFL's Logic Synthesis Library [6] to translate the intermediate representation (a network of

lookup tables) to an AIG. Additionally, I chose to use Brayton and Mischenko's ABC software [2]. ABC provides visual depictions of AIGs and can refactor (simplify) them.

1.3 Neural Networks Considered

As will be described in chapter 2, neural networks have many forms. This thesis will only consider a subset of all possible neural networks.

Neural networks allowed The intended neural network is a feed-forward, sequentially layered, fully connected, multi-layer perceptron (a traditional neural network). Meaning if and only if some node A is in a layer i (a hidden layer), then each node in layer i-1 has a directed edge ending at A and A begins a directed edge to a node if and only if the node is in layer i+1.

Formal definition Let the transition set be δ and each sequential layer be expressed as L_i. Then this algorithm allows any neural network with the property

$$i \in L_{l-1}, j \in L_l \longleftrightarrow (i \to j) \subseteq \delta$$

for any output or hidden layer L_l.

If a neural network meets the requirements except for being fully connected, create pseudo-edges with a weight of zero for it to become a fully connected neural network.

1.4 Gist of the algorithm

This section describes how I convert neural networks to a layer of lookup tables (more specifically, sum-of-products (SoPs)).

Let the traversal over the neural network be called $\mathcal{N}$ (algorithm 4), the optimized binary decision tree that approximates activation functions (chapter 2) be $\mathcal{B}$ (section 8.2), and the network of SoPs be $\mathcal{T}$.

Preliminaries First of all, the feature space must be binarized (section 9.2.1) and the neural network must be trained.

Decision tree ($\mathcal{B}$) I found that traditional decision trees that used information gain or Gini impurity were too powerful and computationally expensive than what was required. I present $\mathcal{B}$—a linear and optimized binary decision tree that accurately finds the SoP for a neural node. See section 7.2 and section 7.3 for details on the activation functions $\mathcal{B}$ can approximate. Furthermore, $\mathcal{B}$ had the same predictions as the traditional decision trees for these activation functions (see section 6.4). Since $\mathcal{B}$ creates the network of SoPs $\mathcal{T}$ ad hoc, $\mathcal{B}$ is never stored.

Complexity $\mathcal{B}$ is in $\Omega(n)$ and $O(2^n)$ with an average of $O(2^{0.5n})$ where n is the amount of in-degree weights for the given node (chapter 10). Cook's Theorem states that the Boolean satisfiability problem is NP-complete [7]. Since $\mathcal{B}$ is a type of Boolean satisfiability, placing $\mathcal{B}$ in P space would be an unlikely and phenomenal feat. However, $\mathcal{B}$ still managed to get an impressive average-case complexity.

Parsing ($\mathcal{N}$) $\mathcal{N}$ traverses the neural layers in reverse—from the output node (n) to the first hidden layer (L_1). Traversing L_i before L_{i-1} allows any negligible nodes in L_{i-1} to be skipped entirely. Nodes within a layer are independent of each other and can be naïvely parallelized. Each neural node is only parsed over once (and possibly skipped).

Summary In summary, $\mathcal{N}$ parses over the neural network in reverse and employs $\mathcal{B}$ to find the sum-of-products as needed. Then an AIG is produced.

1.5 Reasoning and results

Comprehensibility Essentially, the esoteric nature of the neural network is due to the vast amount of nodes, weights, summations, multiplications, and activation functions. Since AIGs are simpler and more concise than neural networks, they are relatively comprehensible. In chapter 4, I provide further analysis on comprehensibility considering the overall complexity of the neural network and how interpretable the basis (input names and values) is.

Practicality While, some functions are inherently complex and difficult to understand, this thesis succeeds in abstracting a neural network to an

AIG and provides another tool for machine learning experts to use in debugging, analyzing, and understanding the complex neural networks and the representing data.

Results Sometimes $\mathcal{N}$ (the algorithm that converts a neural network to $\mathcal{T}$) picked up noise from the neural network (such as evolving or tentative relationships). This relates to the memorization versus generalization problem mentioned later (the noise was from the memorization side). However, an intersection of several $\mathcal{N}$ over several neural networks highlighted the most important relationships and elicited generalization (section 9.3).

Part II

Background

Chapter 2

Neural Networks

Traditional neural networks train on a data set containing inputs and classifications (supervised learning). After training, the neural network predicts classifications for new inputs.

Neural networks in machine learning originated from the McCulloch and Pitts paper that modeled how neurons might work [8]. A major milestone of the neural network was overcoming the XOR problem which was solved by backpropagation (discovered by [9] and solved by [10]). See [11] for a survey on the theory of neural networks and [12] for a survey on deep neural network architectures and their applications.

2.1 Structure

A lack of formal definitions for neural networks hindered formal proofs in the neural network's early decades. E. Fiesler gave a flexible hierarchical and a universal mathematical definition to give the field a solid and formal foundation [13]. Below is a summary of his hierarchy with some added research.

- **Topology**
  - **Framework**
    - **Number of clusters** (i.e. layers or slabs). Layers and slabs are both subsets of neurons with the same hierarchical level, however, layers are ordered while slabs need not be. Not enforcing order means that the neurons in a slab can be rearranged.

        - **Number of neurons in each cluster**
    - **Inter-connection scheme**
        - **Connection types**
            - **Interlayer connections** are connections between adjacent layers and are the most common connection type.
            - **Intralayer connections** connect neurons of the same layer. A self-connection is a type of intralayer connection where the neuron connects to itself.
            - **Supralayer connections** connect a neuron to a layer that is neither adjacent nor in the same layer.
        - **Connectivity** defines how dense the connections are among the neurons. A fully interlayered-connected network is commonly shortened as a fully connected network–this thesis will use this shorthand version. A plenary neural network is a neural network with all possible connections in the interlayer, intralayer, and supralayer domains.
        - **Symmetry versus asymmetry**. A symmetric connection uses the same weight for both directions (i.e backpropagation and forward feeding). An asymmetric connection uses one weight for both directions.
        - **Order**. Higher-order connections combine the outputs of several neurons as a single input–usually by multiplication. Specifically, a higher-order connection that combines N outputs is called an N-order connection. Furthermore, the highest-order connection determines the order of the entire network. Most neural networks are first order.

- **Constraints**
    - **Weight value range** is the set of values an output may be amplified or diminished by.
    - **Local threshold value range** holds the values allowed for an offset for each neuron's activation. This is often perpetuated with a bias node (neuron) at each hidden layer.
    - **Activation range** is the set of values a neuron may output.

- **Initial state**

  - **Initial weights**. These should be created with random numbers to break symmetry which mitigates the potential null space. Intuitively, allowing the initial weights to be in a substantial subset of the weight value range would further rupture symmetry; however, larger initial weights may result in exploding values during propagation [14].
  - **Initial local thresholds**. The bias nodes can be set with heuristically chosen constants (often the constant equals one). The weights attached to the bias node are chosen as described for initial weights.
  - **Initial activation functions**. Commonly, the initial activation functions are unchanged throughout the training process.

- **Transition functions** update the state of the neural network.
  - **Activations functions** (neuron functions) translate the inputs of a neuron to its respective output given some function a(x).
  - **Learning rules** dictate how weights will be updated.
    * **Cost functions** (also called loss functions or error functions) determine the magnitude of a penalty given a miss. A correct prediction suffers no penalty [14].
    * **Optimizer functions** use the cost found in the cost function to update the weights.
  - **Clamping functions** decide if a neuron will consider incoming information.
  - **Ontogenic functions** change the neural network's topology.

Neural networks for this thesis This thesis will focus on the most common neural networks. Specifically, they will be fully interlayered-connected, symmetric, first-order neural networks with some finite amount of layers and nodes. The networks are trained on a binary data set so binary cross-entropy will be the loss function. The Adam optimizer is used throughout this thesis unless mentioned otherwise. Despite training on a binary data set, the activation functions are real-valued. The types of activation functions recommended are discussed in section 7.3 and section 7.2. The initial weights are randomly initialized to small values. Bias nodes with a value of one are

used in every layer except the output layer unless stated otherwise. Activation functions will not change over time. Furthermore, the clamping and ontogenic functions will not be considered.

Feedforward Some examples in this thesis use matrix notation to describe part of the feedforward process. Simply, $\vec{x}$ is the vector of inputs and $\vec{w}$ is the vector of weights. Furthermore, $\vec{x} \cdot \vec{w}^T = \theta$ where θ is the dependent variable for some activation function f. For example, when computing the activation function $f(\theta)$ for H_1 in fig. 2.1, $\vec{w} = (X_1 \rightarrow H_1, X_2 \rightarrow H_1, bias\_node \rightarrow H_1) = (4.4638, -4.4073, -2.6025)$ and $\vec{x} = (X_1, X_2, 1)$. So if $X_1 = X_2 = 0$, then $\theta = -2.6025$ and H_1 will output $f(-2.6025)$.

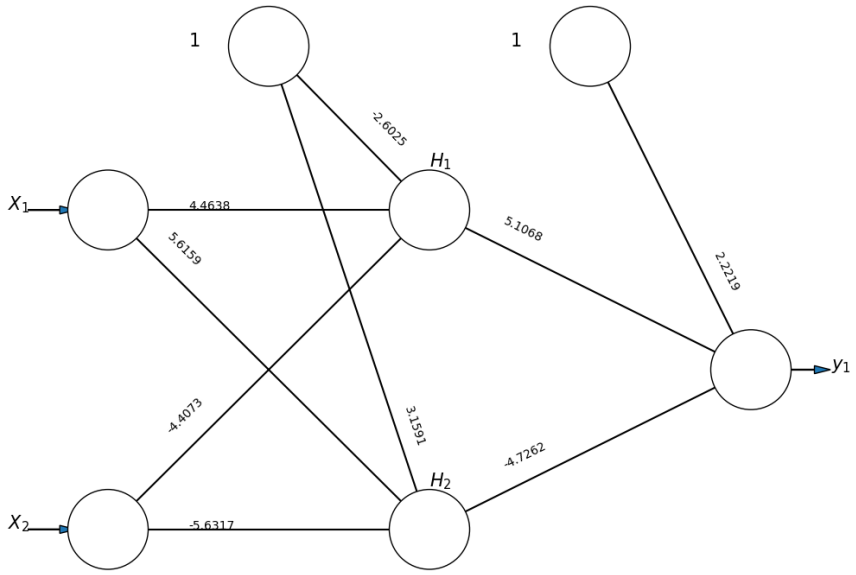
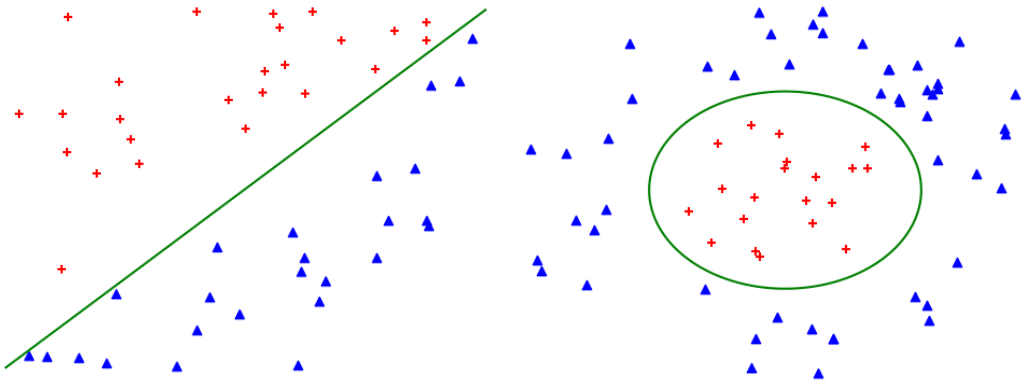

1
1
-2.6025
H_1
X_1
4.4638
5.6159
5.1068
2.2219
y_1
-4.4073
3.1591
-4.7262
H_2
X_2
-5.6317

Figure 2.1

2.2 Types of Activation Functions

Identity function The identity or linear activation function is $f(x) = cx$ where c is a predefined constant. If a neural network is filled with identity activation functions, it can only discriminate linearly separable data (fig. 2.2a gives an example). The linear function is sometimes used for the output neuron to allow a prediction of any real number (given the constraints of the system).



(a) The two classes are linearly separable by $y = x$.

(b) Two classes that are not linearly separable.

Figure 2.2

ReLU The Rectified Linear Unit function (ReLU) is $f(x) = \max(0, x)$. ReLU is the simplest nonlinear activation function and is commonly used in deep learning for its simple derivatives [14]. To remedy the loss of information in the max(0,x) cast, leaky ReLUs ($f(x) = \max(cx, x)$ for $0 < c < 1$) are a similar function that still allow negative values to propagate. Swish is a differentiable function similar to the ReLU [15].

Sigmoids Sigmoid functions take the shape of an "s", and are commonly used for binary classification. Examples of sigmoid activation functions and their output range ($\mathcal{R}$) include

- $f(x) = \frac{1}{1+e^{-x}}$. $\mathcal{R} \in (0, 1)$, exclusively.
- $f(x) = \tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$. $\mathcal{R} \in (-1, 1)$, exclusively.

- $f(x) = 1.7159 \tanh(\frac{2x}{3})$. $\mathcal{R} \in (-1.7159, 1.7159)$, exclusively.

Since the hyperbolic tangent (tanh) can be computationally expensive, LeCun et al. recommend using $f(x) = 1.7159 \tanh(\frac{2x}{3})$ which allows for an approximation with a ratio of polynomials [16]. Symmetric or odd sigmoids (sigmoid functions that do not change when reflected across the line $y = -x$) often converge faster than the logistic function $f(x) = \frac{1}{1+e^{-x}}$ [17]. This is partially due to symmetric sigmoids allowing negative activations. In neuroscience, positive and negative activations are called inhibitory and excitatory, respectively. As negative activations can help neural networks, inhibitory neurons help the human cortex [18].

Softmax While sigmoid functions can be described as a probability distribution over a binary classification, softmax activations give a probability distribution over n classifications. Essentially, each class is given a log probability similar to the logistic function above. Each log probability is then normalized by dividing the log probability by the sum of all log probabilities. By performing normalization, all probabilities in the vector sum up to one [14] (see algorithm 1 for an algorithmic description). After the softmax is computed, the input with the highest probability is predicted.

Algorithm 1: The softmax function.

Input: $\vec{a}$, the input array.
Output: P, an array of probabilities.
forall $a_i \in a$ **do**
    $P[i] \leftarrow \frac{e^{a_i}}{\sum_{a_j \in a} e^{a_j}}$;
return P

Summary of common activation functions

$$\text{ReLU: } f(x) = \begin{cases} x, & \text{if } x > 0, \\ 0, & \text{if } x \leq 0. \end{cases}$$

$$\text{Leaky ReLU: } f(x) = \begin{cases} x, & \text{if } x > 0, \\ cx, & \text{if } x \leq 0. \end{cases}$$

$$\text{Swish} : f(x) = \frac{x}{1 + e^{-x}}$$

$$\text{Logistic: } f(x) = \frac{1}{1 + e^{-x}}$$

$$\tanh\text{: } f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

$$\text{Softmax: } \max(P) \text{ for } P(a_i \in Z) = \frac{e^{a_i}}{\sum_{a_j \in Z} e^{a_j}}$$

Since Keras allows many of these activation functions, custom activation functions, and granular control and inspection of neural networks, Keras (with the Tensorflow backend) is used to create neural networks throughout this thesis. which also allows custom activation functions. More research on activation functions can be found in [19].

Chapter 3

Graphs and Languages

3.1 Graphs

Graph Graphs are abstract data types from graph theory that are defined as the pair (V, E), where

- V is a set of vertices (or states).
- E is a set of edges (transitions between vertices). Edges may be directed (one-way) or undirected (two-way).

Circuits Circuits are specialized types of graphs where each node is called a gate and computes some function. Formally, a circuit is a triple (M, L, G), where

- M is a set of values.
- L is a set of gate functions (or labels) that maps n-ary input to a single output: $M^i \mapsto M$.
- G is a directed acyclic graph labeled with L.

AIGs And-Inverter Graphs (AIGs) are Boolean circuits with

- $M = \mathbb{B} = \{0, 1\}$.
- $L = \{AND, NOT\}$.

- G is finite.

Despite AIGs being structurally inefficient at a hardware level, AIGs excel in abstractions with their effective manipulations of complex Boolean functions and simple design. This thesis is only concerned with comprehensibility so the abstraction is essential, and any hardware inefficiency is irrelevant. Furthermore, simplification and visualization of AIGs have been implemented in free software such as "ABC: An academic industrial-strength verification tool" [2].

MIGs Formally, Majority-Inverter Graphs (MIGs) are Boolean circuits with

- $M = \mathbb{B} = \{0, 1\}$.
- $L = \{MAJ, NOT\}$, where MAJ is the majority gate.
- G is finite.

MIGs were introduced in 2014 with two basis operations: majority and inversion. According to Amarú et al., a set of five primitive transformations creates a complete axiomatic system. By definition, a complete axiomatic system can prove any valid statement in the theory true or false. Therefore, the five primitive transformations allow a simple traversal of the entire MIG representation space. On average, MIG optimization reduces the logic levels by 18% when compared to AIGs optimized by the state-of-the-art ABC tool [2]. Furthermore, Amaru et al. proved MIGs to be a superset of AIGs [20].

While the reduced levels of MIGs would make the graph more concise than AIGs, majority gates are not as intuitive as and/or gates to some people.

AOIGs And-Or-Inverter Graphs (AOIGs) are Boolean circuits with

- $M = \mathbb{B} = \{0, 1\}$.
- $L = \{AND, OR, NOT\}$.
- G is finite.

AIGs use De Morgan's law

$$\neg(\neg a \wedge \neg b) = a \vee b$$

to illustrate an OR. While AIGs may require more NOTs than an AOIG, the optimized AOIG has the same amount of nodes as an optimized AIG for any Boolean formula.

Comprehensibility As discussed in chapter 4, comprehensibility can be defined as conciseness. From this, it stands to reason that MIGs are more comprehensible than AIGs. Furthermore, conciseness can be improved by converting subgraphs in the MIG to OR or AND nodes if doing so does not increase the total amount of nodes. Future work can investigate this Majority-And-Or-Inverter Graph further. See Amarú et al.'s work for information on how conversion may be done [20].

BDDs Binary Decision Diagrams (BDDs) are rooted, directed, acyclic graphs that consist of terminal and binary decision nodes (see fig. 3.1). BDDs were used by Shi et al. to illustrate activation functions that were converted to step functions [1]. Future work may investigate if BDDs are more comprehensible than AIGs.

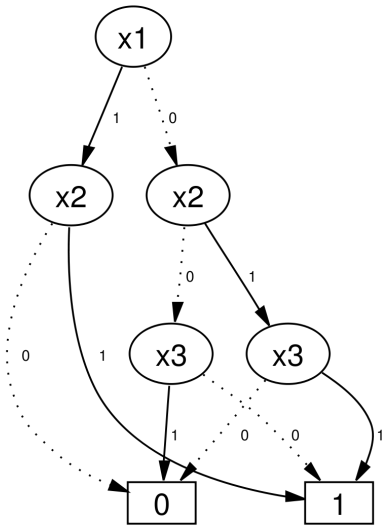


Figure 3.1: A binary decision diagram (BDD) for $f(x_1, x_2, x_3) = \neg(x_1x_2x_3) + x_1x_2 + x_2x_3$. This picture is from Wikipedia.

3.2 ABC

This thesis uses ABC software [2] to illustrate and simplify AIGs. See fig. 3.2 for an example and below for the legend.

Legend for ABC's AIG

- Inputs: up-right triangles ($\triangle$).
- Outputs: upside-down triangles ($\triangledown$).
- AND gates: oval nodes ($\circ$).
  - Labels: ABC labels nodes numerically; the numbers have no logical value.
- NOT gates: a dashed arrow ($\dashrightarrow$).

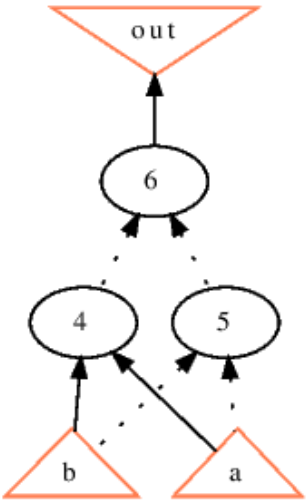


Figure 3.2: An AIG for the XOR function: out $= \neg ab \vee \neg ba$.

- Transitions: an arrow that is solid for transitions without a NOT gate ($\rightarrow$) and dashed for transitions with a NOT gate ($\dashrightarrow$).

ABC commands ABC has a simple command-line integration with the "-c" option—example: "abc -c 'commands_here'". I chose the BENCH [21] format since it produced the smallest file size of the intersection between what EPFL's library [6] could write and [2] could read. After the BENCH file was written, I ran a bash script with the below command to read the BENCH file, translate it into an AIG, refactor (simplify) the AIG, and then finally illustrate the AIG.

```
abc -c ‘‘read ${file}; strash; refactor; show;"
```

Chapter 4

Comprehension

In this chapter, comprehension is analyzed and quantified. Later, I show that an AIG is more comprehensible than a neural network due to its conciseness (fewer nodes, fewer transitions, and simpler functions). Finally, some tips on understanding the AIG are given.

4.1 Understanding

Complete understanding Comprehension is a loaded term, yet the strongest form of comprehension can be expressed well by parts of Descartes' "Rules for the Direction of the Mind" [22]. Essentially, one must intuit each basic fact (input) before understanding the relations among the inputs. Furthermore, this extends to relationships of relationships: one must intuit the first-order relationships before understanding second-order relationships. Descartes' philosophy on comprehension is a bottom-up approach (start from the primitives and work to the more abstract).

Note: In formal logic, first-order logic is defined as quantifying over propositional logic with the existential or universal quantifiers. However, this thesis defines first-order relationships as the innermost relationship to be understood, and n^{th}-order relationships are relationships of the $(n-1)^{th}$-order ones.

Example:

$$\phi = (a \wedge b) \wedge (\neg c \wedge b)$$

Here, the zeroth-order relationships are the atomic values a, b, and c. First-order relationships are $a \wedge b$ and $\neg c \wedge b$. The entire property, ϕ, is a second-order relationship. This definition is not strict. For example, one could consider the NOT operator as another level; I just use n^{th}-order relationships to loosely articulate how nested a relationship is.

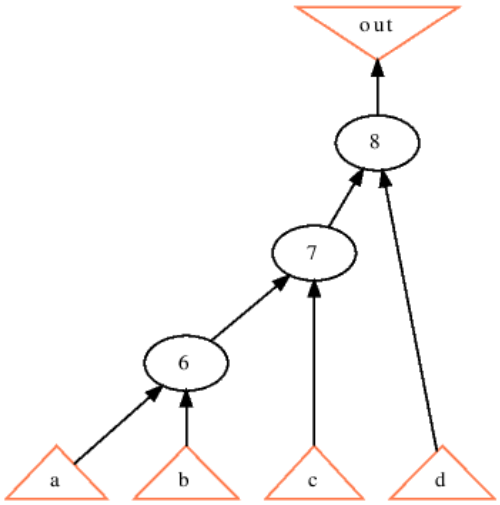
General understanding While a complete understanding is optimal, a disciplined intuition may be impractical in cases with a high-order relationship. For example, consider the unary operator "meta". "meta" transcends its operand using self-reference so "cognition" (thinking) would become "meta-cognition" (thinking about thinking). Extending this concept to n-order meta-cognition (thinking about (n-1)-order meta-cognition), it is time-consuming to intuit each lower-order meta-cognition, but our understandings of "cognition" and "meta" give us a general feel for what an n-order meta-cognition would mean. This arching principle (or pattern) is articulated using mathematical induction in section 4.1.

1. Basis step: The initial (first-order) state of n-order meta-cognition is meta-cognition. Since cognition (zeroth-order) means thinking and "meta" is a self-referential operator, meta-cognition means to think about thinking.
2. Induction step: N^{th}-order meta-cognition is thinking about $(N-1)^{th}$-order meta-cognition.

Figure 4.1: Understanding n-order meta-cognition using a type of "arching principle" called mathematical induction.

AIG example Consider a Boolean function that is the conjunction of all its atoms (example in fig. 4.2a). For people that have previously intuited the conditional of universal truth (i.e. all elements must be true), they can quickly understand this graph. For others, the graph can be understood using mathematical induction (see fig. 4.2b).

Another example Consider the property ϕ over the atoms fight (to contend in physical conflict) and flight (to flee from physical conflict):



(a) AIG for $out = a \wedge b \wedge c \wedge d$.

1. Basis step: a must be true and the first-order relationship $a \wedge b$ must be true.
2. Induction step: traversing the graph, the $(n-1)^{th}$-order relationship and the n^{th}-order relationship must be true.

(b) Understanding fig. 4.2a by mathematical induction.

Figure 4.2: Understanding an AIG that represents the universal truth conditional.

$$\phi = (\text{fight} \wedge \neg\text{flight}) \vee (\neg\text{fight} \wedge \text{flight})$$

After reading the inputs' definitions, start by understanding the innermost relations.

- ¬flight must mean "to not flee from physical conflict."
- ¬fight must mean "to not contend in physical conflict."

Next, we intuit the two conjunctions.

- (fight ∧ ¬flight): to contend in physical conflict and to not flee from physical conflict.
- (¬fight ∧ flight): to not contend in physical conflict and to flee from physical conflict.

Finally, apply the final disjunction and understand ϕ is satisfied in one of two scenarios: 1) where one fights and does not flee, or 2) where one does not fight and flees.

4.2 Quantifying Comprehensibility

This section begins with a basic definition of comprehensibility and extends that definition to allow the comprehensibility of graphs to be determined without human subjects (conditionally). Later, the comprehensibility of a neural network is contrasted with the AIG.

Basic definition: comprehensibility scales by the inverse of the average time spent understanding it (e.g. $\frac{1}{1 \text{ second}}$ is larger and more comprehensible than $\frac{1}{100 \text{ seconds}}$).

Conditional conclusion: If each input, relation, and output of two graphs take approximately equal time to intuit, then the graph with more inputs, relations, and outputs is less comprehensible. Additionally, if the graph with more inputs, relations, and outputs also takes longer to intuit each part, then this graph is still less comprehensible than the other.

Derived definition This definition allows us to quickly determine how understandable an AIG is to another. While I will not explicitly calculate the formula in every AIG, know that when I contrast the comprehensibility of graphs that satisfy the above condition, I will use the derived definition

$$\text{Comprehensibility} = \frac{1}{\|I\| + \|R\| + \|O\|}$$

where $\|I\| + \|R\| + \|O\|$ is the sum of the magnitudes of inputs, relations, and outputs.

Future Work If one takes Descartes' bottom-up approach, then the number of nodes and transitions should convey comprehensibility as seen in the derived definition. However, recall from fig. 4.2a that a top-down approach may be quicker than and elicit approximately the same understanding as a bottom-up approach. I will leave ideas such as this as future work for a comprehensive quantification of comprehension.

AIG versus Neural Network If one were to attempt intuition of a traditional and neural network, each activation function would be seen as a classifier with some real-valued range for some weighted inputs. Analogous

to earlier terms, each neural node in the first hidden layer is a first-order relationship and each node in the n^{th} layer is an n^{th}-order relationship. For hidden sigmoid activation functions, this relationship is the probability that the given set of weighted $(n-1)^{th}$-order relations indicate some intermediary binary classification.

Furthermore, intuiting a trained neural network would require people to intuit each activation function, $(n-1)^{th}$-order relation, each bias constant and weight vector, multiplication, and addition over the real numbers. In contrast, AIGs have fewer distractions and simpler gates—people must simply understand the inputs, AND-gates, and NOT-gates over Boolean values. $\mathcal{B}$ (section 8.2.2) optimizes away negligible inputs and the bias constant so $\mathcal{B}$ produces an AIG with strictly fewer relations and simpler transitions.

See section 3.1 and chapter 2 for the definitions of AIGs and neural networks respectively.

4.3 Inputs

Understanding inputs Consider if fig. 4.2a stood for some fictional female's (Ashley's) preference in men where (a, b, c, d) stood for (confident, humble, masculine, gentle). To cultures where masculine is antonymous with gentle or for people who believe that confidence cannot coexist with humility, fig. 4.2a would make no sense to them despite its simplicity. In this case, no structure can help the individual understand what is being articulated—the inputs must be better defined or understood.

Reducing inputs Additionally, Ashley may have considered confidence to be a trait of masculinity ($\therefore$ confidence $\subseteq$ masculinity). Now, the input basis is not optimized for comprehension (it is not minimized) since the information was duplicated in the confidence and masculinity axes. Future work can investigate the optimal ways to reduce the input space while remaining comprehensible.

Dimension-reduction While there exist dimension reduction techniques such as PCA (see fig. 4.3) that reduce the input space, these techniques may not be facilely understandable. For example, PCA casts m inputs into n components ($PCA : i^m \mapsto c^n$ for $m \geq n$) by combining information from inputs

and omitting other information. Each component in the component-space is likely to be more difficult to understand than each input.

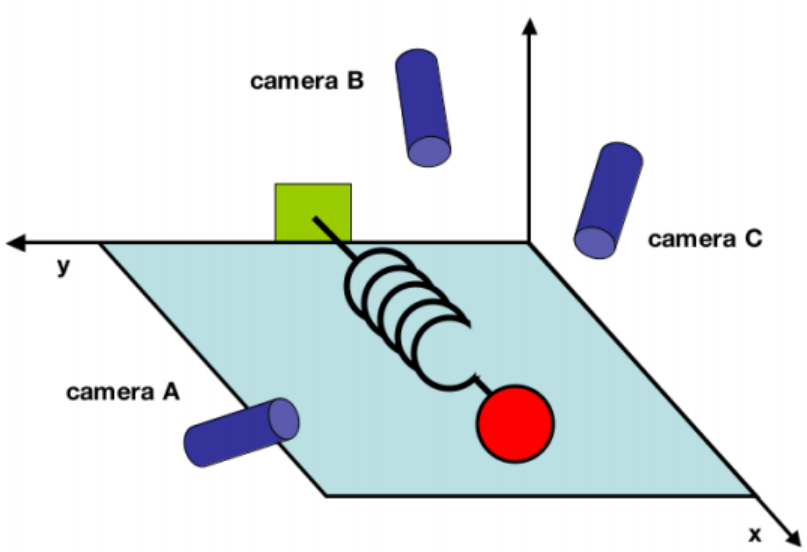


Figure 4.3: Shlen's PCA tutorial used this example of data duplication [23]. Only one camera is needed to predict the spring's position at any given time, yet multiple cameras are giving us variants of the same data. Dimension-reduction techniques such as PCA help alleviate duplicated information such as this.

Many inputs

While comprehension is possible by unconscious or conscious means, both ways are impractical for a human to learn large data sets or intricate graphs.

Unconscious Consider MNIST [24]: this problem is to predict hand-written digits using numerous pixel values. One might reason that since humans can decipher digits given by dots, people can also interpret a digit from a series of numeric dot values. As shown by BrainPort (a computer to brain interface that sends signals to a blind person's tongue from a camera allowing them to see [26]), it may be possible for our unconscious minds to reorganize the relationships seen by numeric pixel values to visualize a digit, but for the sake of

facile comprehension, this is eccentric and difficult [25,26]. Furthermore, cognitive psychologists have coined this type of learning as "implicit learning" which our conscious mind cannot fully acknowledge nor articulate [27]. Even if one could intuit large data sets this way, the person could not articulate the findings to another.

Conscious As for our conscious brain, we can only hold a small, finite amount of information in our working memory at a time [28]. Intuiting a concept means that the concept is stored into a longer-term portion of our memory from our working memory, and this process takes time for each concept to be stored. Therefore, there exist concepts with such intricacy that will take too much time to be realistically intuited.

Patterns However, a multitude of inputs, relations, and outputs does not necessitate complexity. If one can recognize groups or patterns in the mass of information, some understanding can be elucidated. For example, the numerous pixels in the MNIST input space can be understood by aligning shaded pixels (not numeric values) as the original images. Then one can correlate lines and curves from the original data to pixel values in the data set.

Patterns via machine learning Alternatively, one can employ a convolutional neural network (CNN) to highlight patterns of massive data. Given many inputs, CNNs learn patterns independent of location and create a feature-map of these patterns. One can then learn which features are associated with which types of curves and/or lines. Tools such as saliency masks can help humans understand the patterns learned [4].

4.4 Other

Tips Understanding the entire AIG may still be a daunting task–even taken one relationship at a time. For the more complex AIGs, it may be better to consider individual sequences as they come about. Realize that a zero input to AND forces that gate to become zero, no matter what the other input is. Since this other variable is considered a "don't care", you can skip all nodes which create the "don't care" variable for this input sequence. See theorem 9.2.2 for an example.

Final notes The user must understand that the AIG model is an abstraction of the neural network that approximates the possibly biased data set. According to Descartes', believing in false relationships makes one less knowledgeable than if one did not know any relationship exists [22].

Further reading Comprehension is at the intersection of philosophy, cognitive psychology, neuroscience, and more. Furthermore, machine learning and artificial intelligence attempt to capture comprehension using mathematical models. For those interested in truth and axioms, I recommend reading Tarski's theories of truth and satisfiability [29] and Gödel's incompleteness theorems [30]. While Tarski argued that truth is problematic, but the satisfaction of properties are obtainable, this thesis uses truth as defined in a Boolean context. For philosophies of comprehension and cognitive systems, there exists associationism [31] and connectionism [32, 33] among many others. For more research into self-referential logic and paradoxes, investigate Alan Turing's work in [34]. To research interactions and strategies among two or more agents, read into game theory [35–37].

Chapter 5

Related work

5.1 AIGs and LogicNet

In 2020, an article was published articulating how a group of researchers converted a neural network to and-inverter graphs. Shown in figure 5.1 are the three pipelines contrasted in [3].

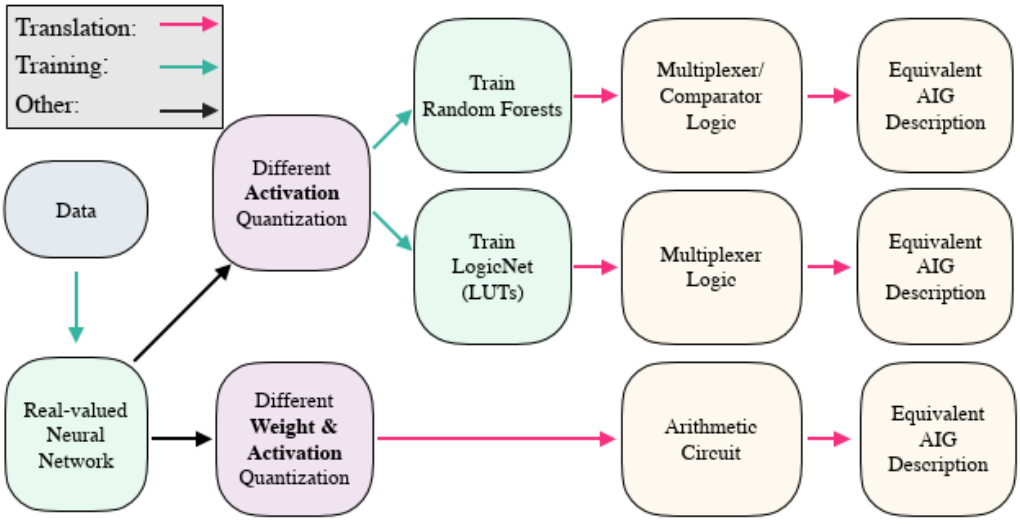


Figure 5.1: The pipelines contrasted in [3].

In subsection 4.3 of [3], the researchers took each input for each activation node and trained a separate decision tree for each bit of each output. While

this technique seems functional, it also seems inefficient. Chatterjee created a network of lookup tables (LUTs) eliciting generalization, and each lookup table in the network was trained on the input from the layer prior and the final output. [3] referred to Chatterjee's network of LUTs as LogicNet.

$\mathcal{N}$ took the idea of global training from LogicNet when parsing the neural network in reverse algorithm 4. This reverse parsing allowed $\mathcal{N}$ to skip negligible neural nodes and to prevent islands or deadends in the AIG. Furthermore, $\mathcal{B}$ (an optimized, binary decision tree) employed a local training technique by approximating each activation function as a sum-of-products (SoPs). Finally, $\mathcal{N}$ produces a network of SoPs similar to LogicNet.

Survey of "Making Logic Learnable with Neural Networks" by Brudermueller et al.

Introduction This paper combines the learning ability of neural nets with the interpretable and formally verifiable logic circuits. Brudermueller et al. begin with a trained neural network then convert the neural network to random forests or multi-level look-up tables (LUTs), and then to AND-Inverter graphs (AIGs). The researchers discovered that direct conversion from NN to AIGs does not create easily interpretable models and has a drastic loss of accuracy. The intermediate representation (IR) of random forests and LUTs elicited generalization in the final output.

There is very little research on logic operations being used to generalize. This paper points to "Learning and Memorization" (Chatterjee, 2018) as the only known example (Chatterjee used factoring to generalize). These simplified structures allow one to test the sensitivity of the inputs.

Background

SAT The Boolean satisfiability problem (SAT) asks if there exists a set of inputs (given some restraints) that invokes a true output. Public SAT solvers exist with O(n) complexity–where n equals the number of variables [38–40].

AIG The idea to use an AIG as the comprehensible output came from Brudermueller et al. Furthermore, all Boolean functions can be represented by AIGs. Fortunately, freely-available software called "ABC" can take AIGs

as input and provides powerful transformations such as redundancy removal to make the AIG more comprehensible. Additionally, ABC includes an SAT solver called MiniSAT for future work to test properties on neural networks [2].

LogicNet LUTs (lookup tables) save previously calculated results in an array-structure. An N-bit LUT encodes a Boolean function with a fan-in of N with 2^N entries. LogicNet (Chatterjee, 2018) used LUTs in successive layers like an inter-layer-connected neural network. Each LUT in a layer receives inputs from a few LUTs in the previous layer–these connections are chosen at random. Chatterjee describes this as a memorization process with noise as opposed to a backpropagation process. Each LUT in LogicNet is trained on the output of the entire network.

Related work

- Chatterjee and Mishcenko illustrated how circuit-based simulations can detect overfitting in [41].
- The researchers in [42] used a deep neural network to learn Boolean satisfiability as an alternative to SAT solvers.
- In [43], binarized weights and activations in neural networks were used as a precursor to hardware compilation.
- Murdoch et al. define interpretability as "the use of machine-learning models for the extraction of relevant knowledge about domain relationships contained in data." [44].

Pipelines tested in Brudermueller et al. [3]

Key notes from neural network → random forests → circuit → AIG

- Takes sets of activations from the trained neural network.
- Creates multiple data sets to train multiple random forests on.
- A trained neural network can generalize while general logic synthesis memorizes the specific inputs.

- Random forests are implemented to approximate don't-care minimization (some inputs do not affect the output).
- The don't-care-based dependency elimination is a simple step to abstracting the formula. Example: $ab + a\neg b \rightarrow a$.

Key notes from neural network → LogicNets → circuit →AIG

- Similar to the random forests pipeline, the training data is created from the activations of the neural network.
- Logicnet is beneficial since it contains layers of factorized LUTs that generalize over simple LUTs by pulling common factors out of the training data [41].
- In contrast to Logicnet, most circuits are a straightforward sum-of-products (like is produced by $\mathcal{N}$). The sum-of-products (SOP) circuit lists every known possibility that creates a true output in conjunctions then if one of those conjunctions is true, the circuit outputs true. According to Brudermueller et al., this is similar to listing sample data and is not enumerable nor generalizable. However, chapter 9 shows that the intersection of several of these circuits can show generalization.

Both According to Brudermueller et al., both pipelines have lower complexity (AIG gate count) and improved accuracy compared to a direct NN → circuit transition.

5.2 "On Tractable Representations of Binary Neural Networks" by Shi et al. [1]

In [1], Shi et al. trained a neural network with sigmoid activations which were later transformed into step activations to allow an easier translation to an ordered binary decision diagram (OBDD or BDD). For BDDs, variable ordering is essential to create the most concise BDD. Furthermore, this paper introduced the connection between approximating a neural node and the Bayesian network classifier to an ODD in [45] with $O(2^{0.5n}n)$ complexity. While Shi et al. also converts activation functions to a binary step function based on some threshold, I extend this research by exploring which activation

functions are best suited for the binary function cast in section 7.2. This paper came out after I had completed much of my experiments. If I did my thesis after this paper, it would further extend this work.

Part III

Methods

Part III articulates the work and thoughts that went into compiling an AIG from a neural network. Chapter 6 shows a prototype created in Python using Keras sequential neural models, Sklearn's decision tree, and other Python tools which empirically produced an AIG with a ludicrous $O(5^n)$ time complexity. Chapter 7 recognizes the extensive computation time required to elicit the XORNN prototype and lays the groundwork for an optimized binary decision tree ($\mathcal{B}$) in chapter 8. Chapter 8 also defines how the network is parsed ($\mathcal{N}$). Finally, chapter 9 introduces AIG construction with real data sets–including those with "don't-care" variables. Chapter 10 in part IV gives supporting evidence that $\mathcal{B}$ reduced the worst-case complexity to $O(2^n)$ and the average case to $O(2^{0.5n})$.

Chapter 6

XORNN

This chapter investigates neural networks that can solve the XOR problem as a prototype to construct AIGs. Efficient algorithms and other examples are introduced in chapter 8 and chapter 9 respectively.

6.1 Definitions

XOR definition For the prototype, I used the simple, nonlinear data set XOR. Formally, XOR can be defined as $f(a, b) = (a \vee b) \wedge \neg(ab)$. $f(a, b)$ gives us this truth table for XOR (table 6.1).

a	b	$f(a, b)$
0	0	0
0	1	1
1	0	1
1	1	0

Table 6.1

XOR problem The simplest neural network that can solve the nonlinear, XOR problem poised by [9] requires three layers. The first layer (the input layer) will be composed of two input nodes and one bias node. The second layer (a hidden layer) will have two nodes and one bias node. The final layer

(the output layer) will be a single node. The hidden nodes and the output node will be equipped with the sigmoid activation function.

XORNN defined Let AllXORNN be the set of all neural networks that perfectly fit the XOR data set and have the topology as described above and as visualized in fig. 6.1. An element in AllXORNN will be referred to as XORNN. The data found in table 6.1 will be used to train XORNN. Since all possibilities of the XOR data set are known and trained upon, perfect accuracy is attainable and generalization is not a concern–that is, XORNN cannot be overfitted. However, I found that XORNN's created with the code given in section 6.2.1 often reached a local minimum of 75% accuracy for at least a million epochs. If the neural network had less than 100% accuracy after about 5000 epochs, it was better to simply reset the weights and start again.

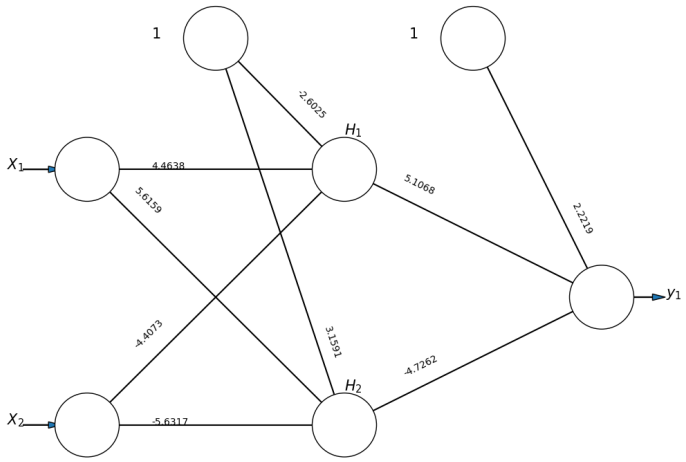


Figure 6.1: Every XORNN has the same topology–only the weights differ.

6.2 Creating XORNNs

This section talks about the highlights of efficiently creating XORNNs. The more XORNNs that can be produced, the better the sample data will be.

6.2.1 XORNN Code

This section explains how a XORNN was created in pseudo-code in algorithm 2 and shows the actual Python script in section 6.2.1. An example of the boundaries drawn by a XORNN is given in fig. 6.2. Section 6.2.2 explains how and why this algorithm was refined.

These Python libraries were used to create XORNN:

- Keras is an API for creating neural networks with the user's choice of

Algorithm 2: Pseudocode for the Python script in section 6.2.1.

Output: XORNN
for $i = 0 \ldots 49$ **do**
  Initialize an XOR neural network (XORNN) with the topology illustrated in fig. 6.1.
  Train XORNN on the XOR data set (fig. 6.6).
  if *Accuracy is 100%* **then**
    return *XORNN*
print("This is very unlikely to be printed.)
return

Theano or Tensorflow for the backend.

- NumPy is advertised as "[t]he fundamental package for scientific computing". NumPy provides high-level functions to manipulate C-style arrays.

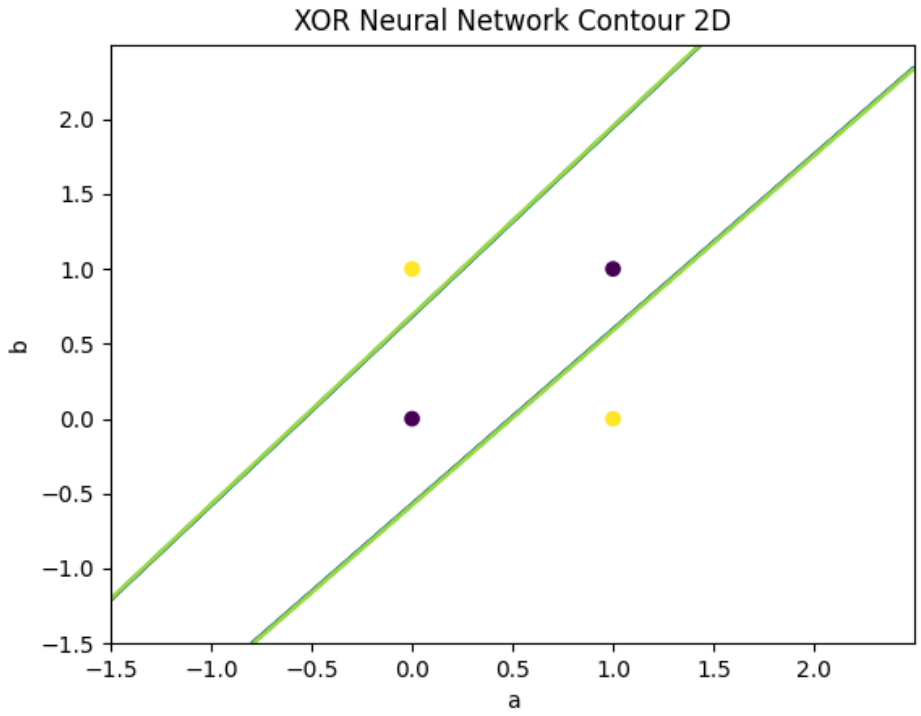
- Scikit-learn (also called sklearn) is a general machine learning library with a broad choice of machine learning algorithms and metrics to measure these algorithms.

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from sklearn import metrics

def generate_xor():
  return np.array([[0,0],[0,1],[1,0],[1,1]]), np.array
      ([0,1,1,0])

def create_xor_nn():
  inputs, outputs = generate_xor()
  #build a new neural net (with new random weights)
      until the neural net can reach 100% accuracy after
       5000 epochs. Times out after 50 tries.
  #Most of the time, three or fewer tries are needed.
  for _ in range(50):
    #first we define our model as sequential
```

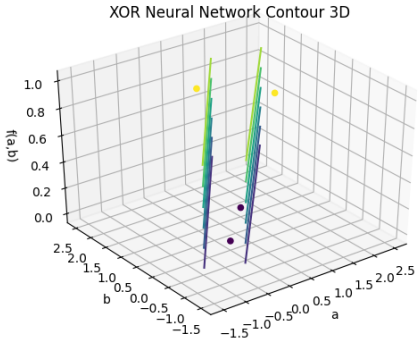
```
    xorModel = Sequential()
    #Dense() is a Keras object for a fully connected
       layer
    #the first add includes the input layer and the
       first hidden layer
    xorModel.add(Dense(2, input_dim=len(inputs[0]),
       activation='sigmoid'))
    xorModel.add(Dense(1, activation='sigmoid'))
    xorModel.compile(
    loss='binary_crossentropy',optimizer='adam',metrics
       =['binary_accuracy']
    )

    #train the neural net
    history = xorModel.fit(inputs, outputs, epochs
       =5000, verbose=0)
    predictions = xorModel.predict_classes(inputs)
    accuracy = metrics.accuracy_score(predictions,
       outputs)
    print("Accuracy: ``, accuracy)
    #stop trying for new neural networks once we
       achieve perfect accuracy
    if accuracy == 1:
      break
  if accuracy != 1:
    print("If everything is functional, it is highly
       unlikely you will see this message.")
    exit(-1)
  return xorModel

xorModel = create_xor_nn()
#once we have the model, we can save it for persistence
    across sessions
saveToThisPath = '/tmp/xor.kerasModel'
xorModel.save(saveToThisPath, overwrite=True,
   include_optimizer=True)
```

If we save our XORNN, we can load the saved model with this script:

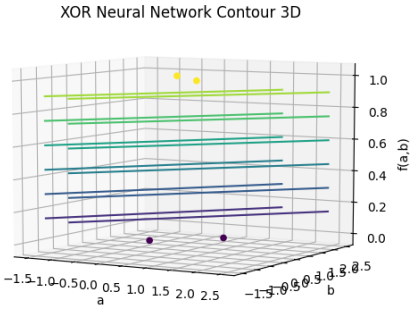
```
import keras
loadedXor = keras.models.load_model('filePathHere')
```



(a) A 2D contour plot of XORNN predictions with the XOR data set plotted. The slope is so steep, all of the contour lines have agglomerated together.



(b) A 3D contour plot of the same XOR neural network in fig. 6.2a.



(c) A side view of fig. 6.2b.

Figure 6.2: Graphs of a XORNN boundary (lines) compared with the XOR data set (dots).

6.2.2 Rationale for Algorithm 2

Local minima As illustrated in fig. 6.3, I discovered that the neural networks with suboptimal accuracy after 5000 epochs would often remain suboptimal for at least 1,000,000 epochs. To remedy being stuck in this local minima, I just reinitialized the neural network with random weights after 5000 epochs. Future work can investigate if there is an epoch threshold where suboptimal NNs will never become a XORNN and the basis of original weights that elicit this phenomenon.

Degeneracy Additionally, it was shown that a few XORNNs' accuracies decreased (therefore losing status as a XORNN) as the epochs surpassed 50,000. 64.4% of all NNs were XORNNs at 50,000 epochs while 100,000 epochs only yielded XORNNs 58.4% of the time. With the Adam optimizer, NNs were consistently found to remain in a local minimum about 42% of the time. Preliminary results indicate that the SGD (stochastic gradient descent) optimizer which Adam was based on also stalls in this local minima. The code in section 6.2.1 was created with this experiment in mind.

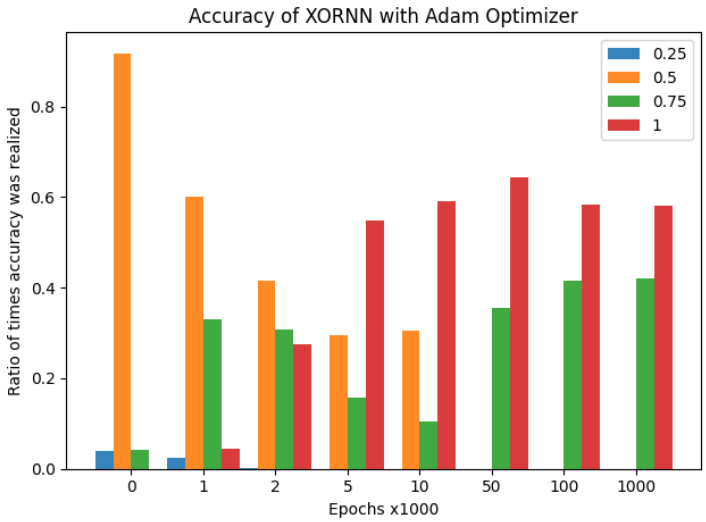
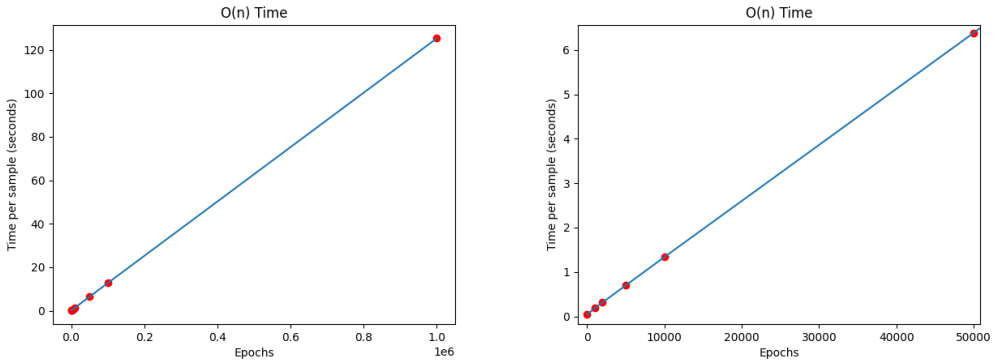


Figure 6.3: Epochs 0 through 10,000 had 500 samples each. 1,000,000 epochs had 100 samples. No sample was reused and tested with the code in section 6.2.1.

6.2.3 Time Creating XORNN

Experiments were done with a six-threaded Intel i5-9600K CPU. To prevent threads from obtaining the same pseudo-random weights, a short lock guarded the critical section which initialized weights with a random seed. This lock and the other required overhead were negligible compared to the training time of the model. Since samples were run in a six-thread parallel, simply multiply the times in the graph by six to estimate the time needed for a single-thread run.



(a) The training time is linear with the number of epochs.

(b) fig. 6.4a zoomed in by omitting the two largest epochs.

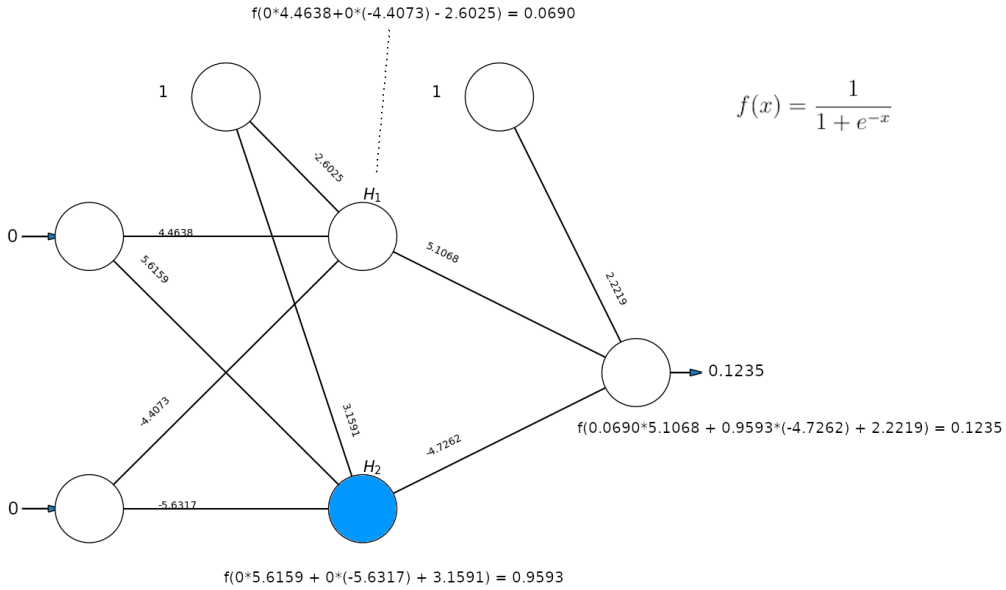
Figure 6.4

GPU training GPU training was not implemented due to the small size of the matrices. GPUs excel at multiplying large matrices which increase proportionally to the number of nodes in the fully connected neural network. XORNN does not have many nodes. In our models for XORNN, a total of seven nodes are used. While implementing processor-parallelism, I found Keras' backend Theano was throttled by Theano's locks; this is why the Tensorflow backend was used instead.

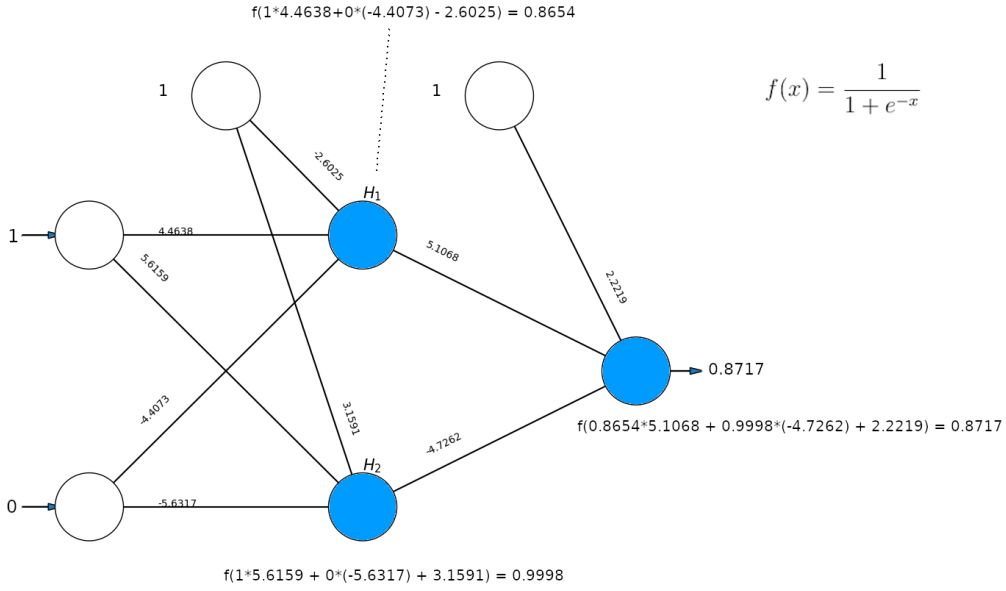
6.3 Understanding XORNN

To gain a better understanding of the rule extraction techniques, let us exhaustively traverse and illustrate the XOR data set with a specific XORNN

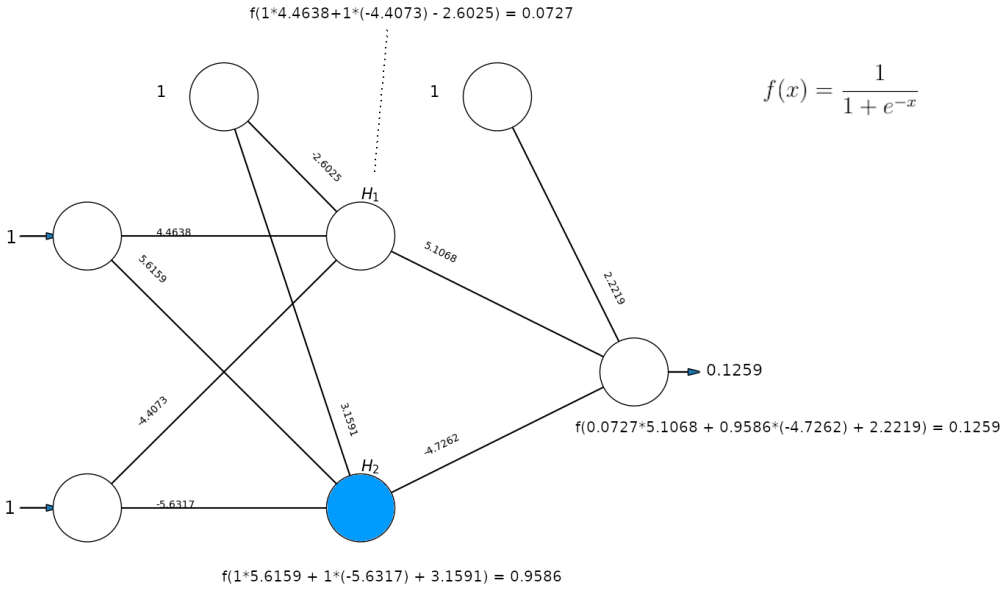
in fig. 6.5. Using the activation function $f(x) = \frac{1}{1+e^{-x}}$, the blue nodes indicate an activation greater than 0.5. Figure 6.6 converts fig. 6.5 into a more comprehensible truth table.



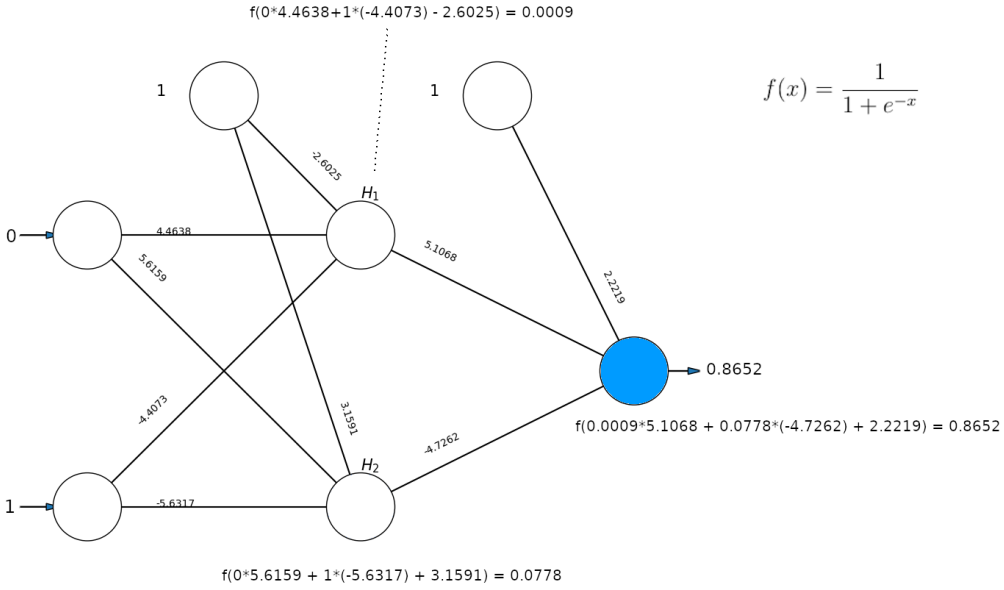
(a) A walkthrough with inputs = {0,0}.



(b) A walkthrough with inputs = {1,0}.



(c) A walkthrough with inputs = {1,1}.



(d) A walkthrough with inputs = {0,1}.

Figure 6.5: Exhaustive walkthroughs of a XORNN.

x_1	x_2	$h_1(x_1, x_2)$	$h_2(x_1, x_2)$	$out(h_1, h_2)$
0	0	0	1	0
1	0	1	1	1
1	1	0	1	0
0	1	0	0	1

Figure 6.6: The logic of the XORNN in fig. 6.5 approximated as an LUT.

6.4 Learning XORNN

This section investigates using a network of decision trees (called XORD) to approximate a XORNN.

6.4.1 Creating a Decision Tree Network

Tree graphs The tree graphs in this section use sklearn's [46] built-in graphing function which uses [47] and several other libraries. Consider the optimal decision tree for the XOR data set as illustrated in fig. 6.7. Each of the parent (white) nodes 5 lines of text. The first line is the conditional of the node that dictates which branch to take (left is true and right is false). The second line shows the Gini impurity of the conditional. The third line represents the number of samples the node was given at this point, and the fourth line shows how many samples of each class were given. Finally, the last line is the tree's best guess of a class given its current path. The colored nodes are the leaf nodes of the tree and signify the end of the path.

Learning Nodes To learn XORNN, a simple decompositional technique is employed: learn each node with a decision tree. Afterward, the decision trees are layered according to their respective nodes (see fig. 6.8). Afterward, the tree is analyzed in section 6.5.

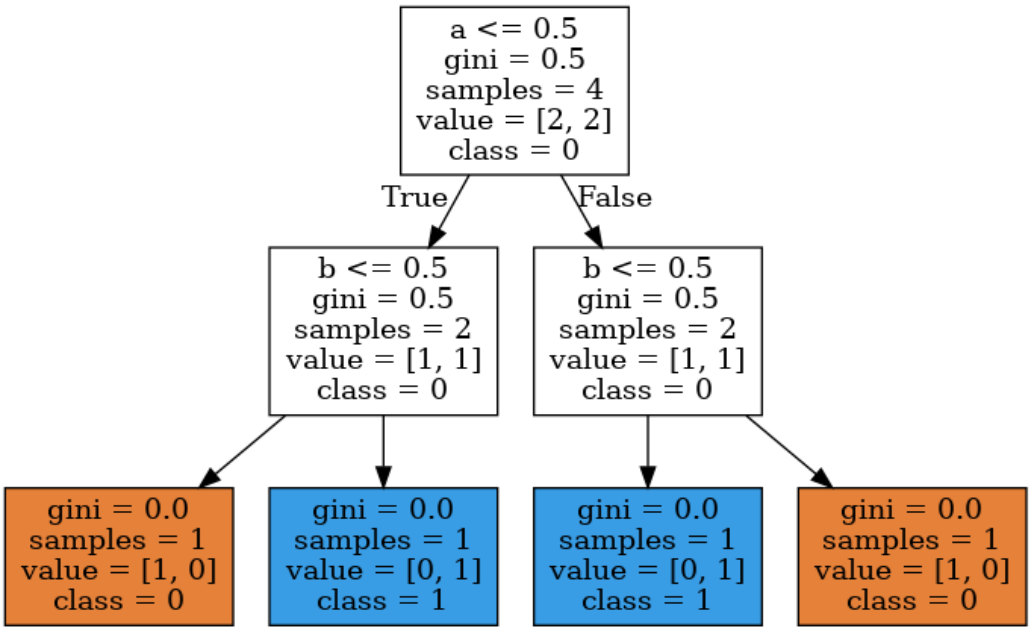


Figure 6.7: The optimal decision tree for the XOR data set created with Sklearn [46].

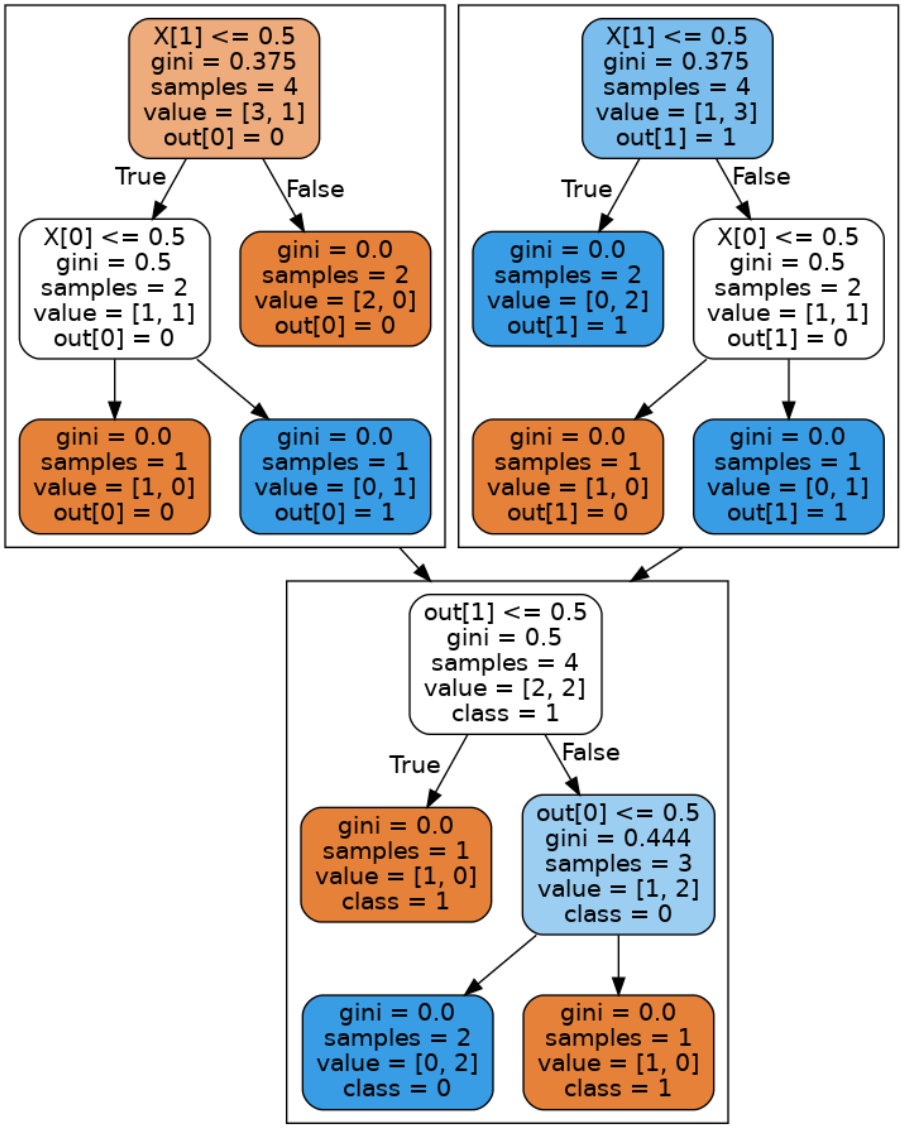


Figure 6.8: A decision tree network learned from the binarized activations of the XORNN pictured here fig. 6.5. The top left decision tree learned the activation from node H_1, the top right decision tree's activation function was from H_2, and the final decision tree at the bottom represents the activation from the output node. As one can see, this decision tree network requires reorganizing to be shown as fig. 6.7.

6.5 Understand the Decision Tree Network

This section translates fig. 6.8 to a Python function by nesting the conditionals of the decision tree. After the Python is tested to correctly model fig. 6.8, I convert the Python to Boolean algebra to prove it correctly approximates XORNN and the XOR data set.

XORD to Python Since all inputs in the decision tree are binary, it is known that "$x[1] <= 0.5$" means "$x[1] == 0$" and "$x[1] > 0.5$" means "$x[1] == 1$". Next, I nest the conditionals in Python.

```
#First we must recognize that since all inputs are
    binarized in the decision tree learning and x is the
     list of inputs, therefore
#x[1] <= 0.5 is equivalent to x[1] == 0.
#x[1] > 0.5 is equivalent to x[1] == 1.

def xord(x):
  #initialize the output node's input
  out = [-1,-1]
  #H1 (Hidden node 1) and top-left decision tree
  #produces x[0] (here we call it out[0]) for
  #the out node
  if x[1] == 0: #gini = 0.375
    if x[0] == 0:  #gini = 0.5
      out[0] = 0 #samples = 1
    else:
      out[0] = 1 #samples = 1
  else:
    out[0] = 0 #samples = 2

  #H2 and top-right decision tree produces out[1]
  if x[1] == 0: #gini = 0.375
    out[1] = 1 #samples = 2
  else:
    if x[0] == 0: #gini = 0.5
      out[1] = 0 #samples = 1
    else:
```

```
        out[1] = 1 #samples = 1

    #Output node and bottom decision tree
    if out[1] == 0: #gini = 0.5
      return 1 #samples = 1
    else:
      if out[0] == 0: #gini = 0.444
        return 0 #samples = 1
      else:
        return 1 #samples = 2

#x can be any of the four data points in XOR
allInputs = [[0,0],[0,1],[1,0],[1,1]]
for x in allInputs:
  print(x, '' returned '', xord(x))
```

The result of running the script in section 6.5:

```
[0, 0]  returned  0
[0, 1]  returned  1
[1, 0]  returned  1
[1, 1]  returned  0
```

Conditionals to SoP Now let's simplify fig. 6.7 further by translating each if-else chain to a sum of products (SoP). Simply begin with the outer-most if, then create a product for each path (following the logic as each conditional). If class 1 is reached during the traversal, write the current product in the sum of products. Any paths that end with a class 0 are omitted. The final SoP will fully encompass the space of our if-else blocks. This idea of only including class 1 in the sum-of-products was tested to work here and implemented in $\mathcal{B}$ in section 8.2.

```
#Change all comparisons and assignments to Boolean
    logic
#Concatenate each if conditional with AND then the
    nested
#if conditional and print the product if an assignment
    of True is found.
```

```
def xord_simplified(x):
        #initialize the output node's input
        out = [False, False]
        #H1 (Hidden node 1) and top-left decision tree
        #produces out[0] for the out node's input
        out[0] = (not x[1] and x[0])
        # if not x[1]: # add not x[1] to our product
        #     if not x[0]: #add not x[0] to the product
        #         out[0] = False
        #     else: #x[0] is true #remove not x[0] and
            add x[0]
        #         out[0] = True #print our first
            product ``not x[1] and x[0]"
        # else: #x[1] is true #remove ``not x[1] and x
            [0]"
        #     out[0] = False

        #begin new empty product with each non-nested
            if block
        #H2 and top-right decision tree produces out[1]
        out[1] = not x[1] or (x[0] and x[1])
        # if not x[1]: #product = not x[1]
        #     out[1] = True #print product. SoP = not x
            [1]
        # else: # x[1] is true #flip the final
            conditional. product = x[1]
        #     if not x[0]: #product = x[1] and not x[0]
        #         out[1] = False
        #     else: # x[0] is true #product = x[1] and
            x[0]
        #         out[1] = True #SoP = not x[1] or (x
            [0] and x[1])

        #Output node and bottom decision tree
        return not out[1] or (out[0] and out[1])
        # if not out[1]:
        #     return True
        # else: #out[1] is true
```

```
#       if not out[0]:
#           return False
#       else: #out[0] is true
#           return True
```

In Python, "and" operators take precedence over "or" operators, so the parentheses are not needed but help with cohesion for those less attenuated to Python's parsing. Without comments, our function is now:

```
def xord_simplified(x):
  out = [False, False]
  out[0] = (not x[1] and x[0])
  out[1] = not x[1] or (x[0] and x[1])
  return not out[1] or (out[0] and out[1])
```

Identical logic With this simplification, we can see that out[1] (node H_2) and the return (output node) share the same simplified logic. Indeed, the ordered weights going into the respective nodes in fig. 6.1 share similar ratios ($3.1591 : 5.6159 : -6.6317 \approx 2.2219 : 5.1068 : -4.7262$). This is due to x being a linear function of its dependent variables which are scaled with their respective weights. This concept is further explored in section 7.2 and brings the idea of identifying equivalent subtrees in section 8.4.2.

Using substitution, our function is defined in a single statement.

```
def xord_simplified(x):
  return (not (not x[1] or (x[0] and x[1]))) or ((not x
      [1] and x[0])
    and (not x[1] or (x[0] and x[1])))
```

This substitution evokes a complicated expression which can be simplified using Boolean algebra to the definition of XOR.

```
def xord_simplified(x):
  return (x[1] and not x[0]) or (not x[1] and x[0])
```

Proving the Python code correct

Theorem 6.5.1. *The Python code* (not (not x[1] or (x[0] and x[1]))) or ((not x[1] and x[0]) and (not x[1] or (x[0] and x[1]))) *is equivalent to* (x[1] and not x[0]) or (not x[1] and x[0]).

Proof. Let x[0] = a and x[1] = b. Then our return statement is:

$$\neg(\neg b + ab) + (\neg ba(\neg b + ab)) \tag{6.1}$$

Change the first product using De Morgan's laws.

$$\neg(\neg b + ab) = b\neg(ab) \tag{6.2}$$

Again, use De Morgan's laws.

$$b\neg(ab) = b(\neg a + \neg b) \tag{6.3}$$

Use the distributivity of AND over OR.

$$b(\neg a + \neg b) = b\neg a + b\neg b \tag{6.4}$$

Use a law of complementation (b and not b must be a false statement).

$$b\neg a + b\neg b = b\neg a + 0 \tag{6.5}$$

Removing the 0 of the logical sum, return the simplified first product.

$$\neg(\neg b + ab) + (\neg ba(\neg b + ab)) = b\neg a + (\neg ba(\neg b + ab)) \tag{6.6}$$

Use the distributive property on the second product.

$$(\neg ba(\neg b + ab)) = \neg b\neg ba + \neg baab \tag{6.7}$$

Remove clear redundancies using the laws of idempotence.

$$\neg b\neg ba + \neg baab = \neg ba + \neg bb \tag{6.8}$$

Use a law of complementation.

$$\neg ba + \neg bb = \neg ba + 0 \tag{6.9}$$

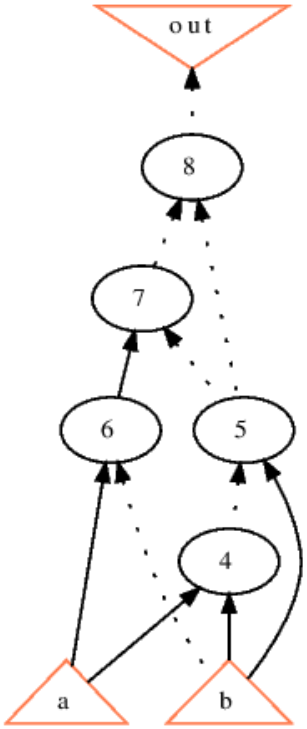
Remove the 0 from the logical sum and return the second product back to the original equation.

$$\neg(\neg b + ab) + (\neg ba(\neg b + ab)) = b\neg a + (\neg ba(\neg b + ab)) \tag{6.10}$$

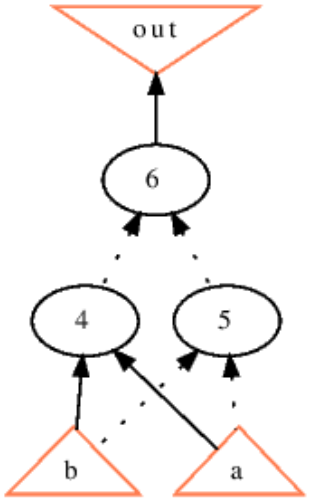
$$= \neg ab + \neg ba \tag{6.11}$$

□

Python to AIG Using AIGER [48, 49] and ABC [2], the logic of the neural network is expressed with an and-inverter graph (AIG) in fig. 6.9. See section 3.2 for details on ABC's AIGs.



(a) AIG representation of XORNN before ABC's simplification.



(b) AIG representation of XORNN after simplification with ABC [2].

Figure 6.9: Equivalent AIG representations of XORNN.

Chapter 7

Importance

The importance of any input scales with the input's influence on class prediction. While importance can be evaluated with Gini impurity or information gain, this chapter finds a more computationally efficient way to determine importance specifically for neural nodes.

Furthermore, this chapter discusses why only weights and some scalar threshold (θ_t) need to be considered in $\mathcal{B}$. Section 7.1 briefly discusses why weights are the "important" factor. Furthermore, section 7.2 and section 7.3 explains the types of activation functions that allow $\mathcal{B}$ never actually compute the function.

Binarization Regarding quantization, I cast all input values (values from the initial data set) into $\{0, 1\}$. This works fine for some inputs, however other inputs have a non-monotonic relationship with the output and too much information is lost with the 0,1 cast. In the latter case, single inputs can be read as two or more inputs which can effectively maintain information (further addressed in section 9.2.2).

7.1 Weights

Each activation function, f has a single dependent variable, θ. θ is found by the dot product of the weight ($\vec{w}$) and input ($\vec{x}$) vectors.

Lemma 7.1.1 and theorem 7.1.2 shows that the importance of each $x_i \in \vec{x}$ is exactly the absolute value of the corresponding $w_i \in \vec{w}$. This is the

discovery that allows $\mathcal{B}$ to skip the computationally expensive information gain or Gini impurity that are often used in decision trees.

Lemma 7.1.1. *For every* $x_i \in \vec{x}$, $\dfrac{d\theta}{dx_i} = w_i$.

Proof.

$$\theta = \vec{x} \cdot \vec{w}^T = w_0 x_0 + w_1 x_1 + \cdots + w_{n-1} x_{n-1} + 1 \times w_b b \tag{7.1}$$

$$\frac{d\theta(\vec{w}, \vec{x})}{dx_i} = w_i \tag{7.2}$$

□

Theorem 7.1.2. *The importance of* $x_i \in \vec{x}$ *is the absolute value of the corresponding* $w_i \in \vec{w}$.

Proof. Since each input ($x_i \in \vec{x}$) has the same range and is scaled by its respective weight (see lemma 7.1.1) in the linear function $\theta(\vec{w}, \vec{x})$, the portion of contribution each x_i has towards θ can be found by taking the ratio of w_i to the absolute sum of all $|w| \in \vec{w}$.

$$importance(x_i) = \frac{|w_i|}{\sum_{w_j \in \vec{w}} |w_j|}$$

Removing the common denominator among all $importance(x_i) \in importance(\vec{x})$ yields:

$$importance(x_i) = |w_i| \tag{7.3}$$

□

Because of theorem 7.1.2, $\mathcal{B}$ sorts all weights in a given node by its absolute value then iterates from the largest (most important) to the smallest (least important).

7.2 Monotonic Functions

This thesis uses a threshold to approximate a real-valued function as a binary function. To binarize a real function into two classes, lemma 7.2.1 can be used. Figure 7.1b provides a visual example.

Lemma 7.2.1. *Given function f and a threshold $\alpha \in \mathcal{R}$*

$$g(\alpha, f(\theta)) = \begin{cases} 1 & f(\theta) > \alpha \\ 0 & f(\theta) < \alpha \\ 0 \vee 1 & f(\theta) = \alpha \end{cases}$$

Alternatively, one can find the independent variable that created in $f(\theta) = \alpha$. Then, any relaxed or strictly monotonic function does not need to compute f for every class decision and instead use lemma 7.2.2. Theorem 7.2.3 proves non-decreasing monotonic functions can use a threshold on the domain, and a similar proof can be made for non-increasing monotonic functions. See section 7.3 for examples of non-monotonic functions that can be used.

Lemma 7.2.2. *A threshold is θ_t in the function:*

$$g(\theta_t, x) = \begin{cases} 1 & x > \theta_t \\ 0 & x < \theta_t \\ 0 \vee 1 & x = \theta_t \end{cases}$$

$0 \vee 1$ *depends on the algorithm. $\mathcal{B}$ takes whichever value renders a quicker computation (a greedy approach).*

Theorem 7.2.3. *For any non-decreasing monotonic function f and any given threshold α on the dependent axis, the corresponding independent value θ_t for $f(\theta_t) = \alpha$ may be used as the threshold without any loss in accuracy. That is, $g(f(\theta_t), f(x)) = g(\theta_t, x)$. g is defined in lemma 7.2.2.*

Proof. By definition, non-decreasing monotonic functions must satisfy the property:

$$\forall(x, y) : x \leq y \longleftrightarrow f(x) \leq f(y) \tag{7.4}$$

Therefore, if $f(\theta_t) = \alpha$, then each case in $g(\alpha, f(x))$ is identical to $g(\theta_t, x)$.

$$f(x) > \alpha \longleftrightarrow x > \theta_t \tag{7.5}$$

$$f(x) < \alpha \longleftrightarrow x < \theta_t \tag{7.6}$$

$$f(x) = \alpha \longleftrightarrow x = \theta_t \tag{7.7}$$

Therefore,

$$g(f(\theta_t), f(x)) = g(\theta_t, x) \tag{7.8}$$

□

Non-increasing monotonic functions Usually activation functions do not satisfy the *non-increasing* monotonic property:

$$\forall(x, y) : x \leq y \longleftrightarrow f(x) \geq f(y)$$

However, if an activation function does, then

$$g(\alpha, f(x)) = \neg g(\theta_t, x))) \tag{7.9}$$

7.2.1 Proving a function is monotonic

To prove a function is monotonic, one can prove its first derivative is always non-positive or always non-negative. An example is given in theorem 7.2.4.

Theorem 7.2.4. *One of the most common activation functions,* $f(\theta) = \frac{1}{1+e^{-\theta}}$*, is a non-decreasing monotonic function.*

Proof. This is done by proving $f(\theta)$'s first derivative is always non-negative.

Using the chain rule, we find $df/d\theta$.

$$f(\theta) = (1 + e^{-\theta})^{-1} \tag{7.10}$$

$$df/d\theta = -(1 + e^{-\theta})^{-2} \cdot -e^{-\theta} \tag{7.11}$$

$$df/d\theta = \frac{e^{-\theta}}{(1 + e^{-\theta})^2} \tag{7.12}$$

$e^{-\theta}$ is positive since e is positive and a real-valued exponent cannot change the positivity of its base nor yield 0 from a non-zero base.

$$e^{-\theta} > 0 \tag{7.13}$$

Any square of nonzero, real-valued numbers must always yield a positive real value. To prove the denominator is nonzero, see that $1 > 0$ and $e^{-\theta} > 0$ by eq. (7.13). Since both values are positive and real, their summation must also be positive and real.

$$(1 + e^{-\theta})^2 > 0 \tag{7.14}$$

Of course, dividing a positive real number by another positive real number results in a positive real number. Therefore,

$$df/d\theta = \frac{e^{-\theta}}{(1 + e^{-\theta})^2} > 0 \geq 0 \tag{7.15}$$

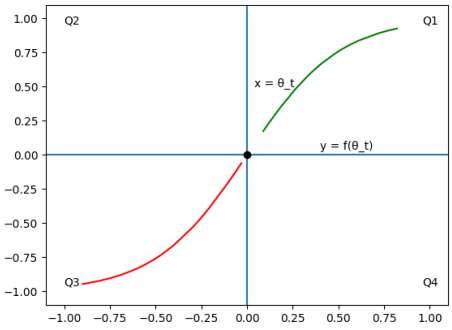
□

7.3 Generalizing to more activation functions

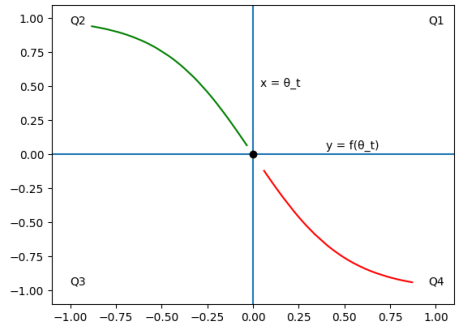
Section 7.2 relies on the monotonicity of the activation function. This section generalizes the principle of using a threshold in the domain instead of the range.

Diagonal quadrants To produce the most accurate results, $\mathcal{B}$ requires a threshold θ_t that when two perpendicular lines (one horizontal and one vertical) are drawn intersecting $(\theta, f(\theta))$, the function only resides on the lines of or inside two diagonal quadrants (see an illustration in fig. 7.1. For monotonic functions, any threshold satisfied this property.

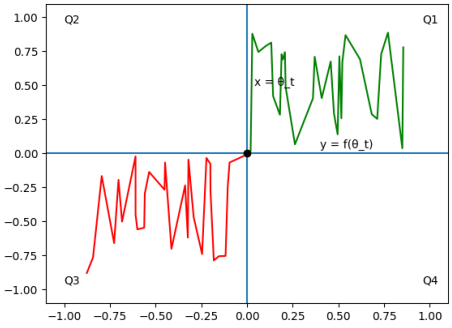
Decent choices Besides satisfying the diagonal-quadrants property, a threshold must also be a "decent" choice. A formal analysis of "decent" choices for the threshold θ_t will be left for future work, but factors such as θ_t being a mean, median, point of inflection, drastic change in slope (ReLU), or an intersection with the y-axis are valid arguments that the threshold is "decent". For odd, monotonic functions, fig. 7.2 shows a good strategy to find suitable thresholds. Figure 7.3 shows examples of finding a decent threshold for functions that are not odd. Finally, fig. 7.4 shows some uncommon activation functions where $\mathcal{B}$ is not guaranteed to do well with.



(a) A function which is in diagonal quadrants Q1 and Q3.

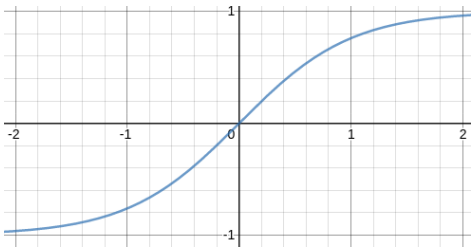


(b) A function which is in diagonal quadrants Q2 and Q4. Notice how the positive class (green) is for values less than θ_t. Accordingly, $\mathcal{B}$ must flip which products (summations of weights) which are considered true and false (see section 7.2).

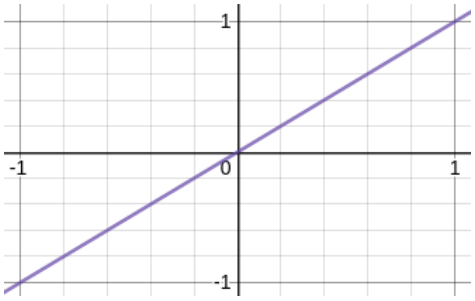


(c) The function does not need to be monotonic.

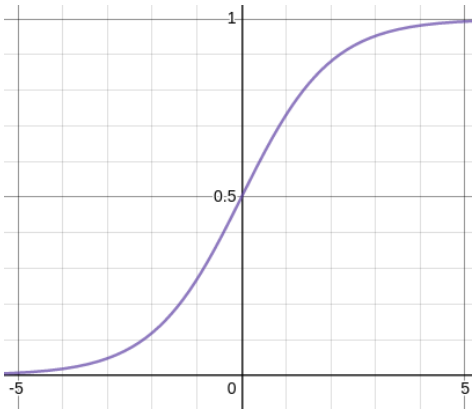
Figure 7.1: Examples of graphs which satisfy the diagonal quadrants property.



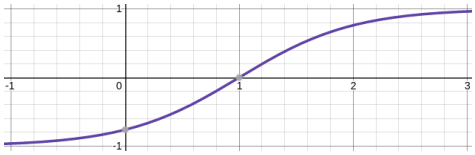
(a) In $\tanh(\theta) = \dfrac{2}{1+e^{-2\theta}} - 1$, $(a, b) = (0, 0)$ so $\theta_t = 0$.



(c) The identity function $f(\theta) = \theta$ has $(a, b) = 0$ and $\theta_t = 0$.

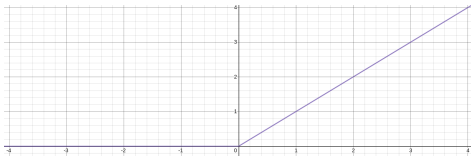


(b) In $\dfrac{1}{1+e^{-\theta}}$, $(a, b) = (0, 0.5)$ so $\theta_t = 0$.

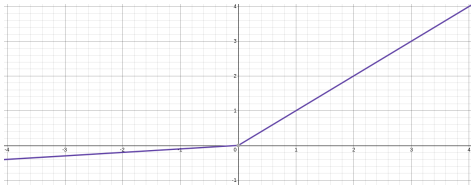


(d) Most activation functions center around $\theta = 0$. This function $\tanh(\theta - 1)$ is a translation of $\tanh(\theta)$ which yields $(a, b) = (1, 0)$ and $\theta_t = 1$. While not recommended as an activation function, this example is to illustrate the uncommon case of $\theta_t \neq 0$.

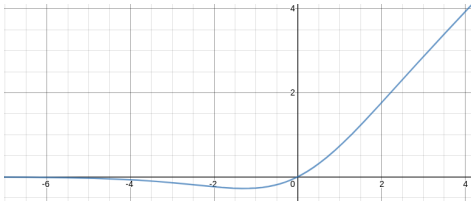
Figure 7.2: If a monotonic function $f(\theta)$ can be translated to an odd function by a constant pair (a, b) in $f(\theta - a) + b$, then $f(\theta)$ has a suitable threshold at $\theta_t = a$.



(a) For the ReLU activation function $f(\theta) = \max(0, x)$, we see a drastic change in slope at $(0, 0)$. Furthermore, ReLU was purposely designed with this sudden change at $\theta = 0$. $\theta_t = 0$ is a quality choice.

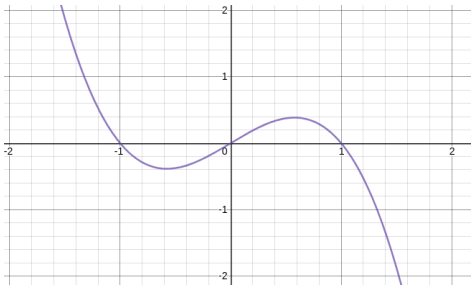


(b) The leaky ReLU activation function $f(\theta) = \max(0.1x, x)$ was made with similar intentions as ReLU. For similar reasons, $\theta_t = 0$ is a decent choice.

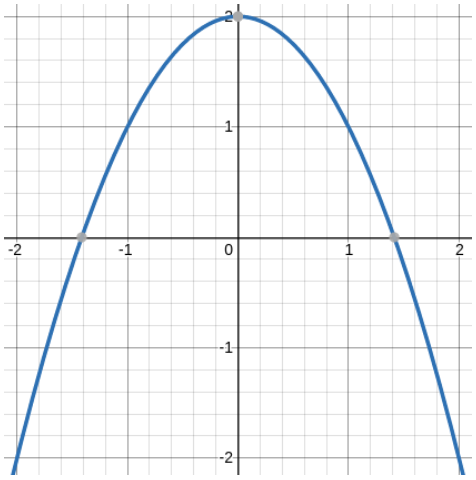


(c) The Swish activation function $f\theta) = \theta \cdot \text{sigmoid}(\beta x)$ models a differentiable ReLU activation function [15]. Like ReLU, Swish has a great choice of $\theta_t = 0$.

Figure 7.3: These functions will not be odd even if translated by any constant pair. However, they still have a decent choice of $\theta_t = 0$ since the function remains one the lines of or in two diagonal quadrants.



(a) $f(\theta) = \theta - \theta^3$. Despite that this function is odd, $\theta_t = 0$ is not a good choice since the function does not remain inside two diagonal quadrants when centered on $\theta_t = 0$. However, if the frequency of weight summations inside $-1 < \theta < 1$ is small, $\mathcal{B}$ can still give accurate results.



(b) $f(\theta) = -\theta^2 + 2$. Even functions have no possible threshold where the function is contained in two quadrants. Therefore, even functions are unsuitable for $\mathcal{B}$.

Figure 7.4: Activation functions where the accuracy of $\mathcal{B}$ will be low or conditional on the weights.

Unsuitable activation functions For uncommon activation functions such as fig. 7.4a, a single threshold is inapposite. While $\mathcal{B}$ can work with any threshold, the results will be less accurate as less of the function is contained within two diagonal quadrants. To regain the lost accuracy, a set of ranges could be implemented such as $\theta_R = \{(-\infty, -1], [0, 1]\}$; however, this would likely make $\mathcal{B}$ more computationally expensive on average. Considering the infrequent use and impracticality of such activation functions on $\mathcal{B}$, I will leave them to future work.

7.4 Application to $\mathcal{B}$

For functions that are contained in quadrants one and three (see fig. 7.1), $\mathcal{B}$ translates weight summations that are greater than θ_t as a product in the sum of products, and weight summations that are less than θ_t are forgotten. For functions contained in quadrants two and four, the opposite can be done.

Algorithm 3 Since each activation function has been binarized, the outputs of previous functions ($\vec{x}$) can be considered a bitmask. Now $\mathcal{B}$ searches for every possible bitmask of $\vec{x}$ whose dot product with the weight vector $\vec{w}$ is greater than the threshold θ_t. Chapter 8 investigates this problem thoroughly.

Algorithm 3:

Input:

- $\vec{w} \in \mathbb{R}^n$, the weight vector.
- θ_t the threshold.

Output: $P \in \mathbb{B}^{m \times n}$ for $0 \leq m \leq 2^n$. P is a table of products that consist of every product in the sum-of-products which represents this given node.

$P \leftarrow \emptyset$

`/* The set` $\mathbb{B}^n$ `can also be described as an n-ary Cartesian power of {0,1}. */`

forall $\vec{x} \in \mathbb{B}^n$ **do**

  if $\vec{x} \cdot \vec{w}^T \geq \theta_t$ **then**

    $P \leftarrow P \cup \vec{x}$

return P

Chapter 8

Parsing the Neural Network

To parse the neural network, each layer must be traversed, then each node in the current layer is converted to a sum-of-products; finally, an AIG is constructed. First, a computationally expensive brute-force attempt is shown in section 8.1. Next, ideas from section 7.2 are implemented into the optimized and novel decision tree ($\mathcal{B}$) which presented in $\mathcal{B}$. $\mathcal{B}$ converts a neural node to a sum-of-products (SoP) ad hoc. Next, algorithm 7 describes the layer parsing. See algorithm 4 for an overview of the neural network parsing and when $\mathcal{B}$ is implemented.

Algorithm 4: A brief algorithmic description of chapter 8.

Input: $N[1 \dots L]$ and θ_t. N is an array representing the layers of a neural network, and each layer is an array of in-degree weights. θ_t is the threshold.

Output: AIG

Let SoP be a two-dimensional table where each element is a sum-of-products.

forall $l_i \in N$ **do**
  forall $n_j \in l_i$ **do**
    $SoP_{i,j} \leftarrow \mathcal{B}(n_j)$

$AIG \leftarrow table\_sop\_to\_aig(SoP)$

return AIG

After the algorithm is explained, supplemental ideas and future work for $\mathcal{B}$ are given in section 8.4. Finally, alternative algorithms to $\mathcal{B}$ are mentioned in section 8.5.

	$w_1 = 3$	$w_2 = -6$	$w_3 = -2$	$w_{act} = \overrightarrow{mask_i} \cdot \vec{w} + 3$	$w_{act} \geq 0$
$\overrightarrow{mask_0}$	0	0	0	3	1
$\overrightarrow{mask_1}$	0	0	1	1	1
$\overrightarrow{mask_2}$	0	1	0	-3	0
$\overrightarrow{mask_3}$	0	1	1	-5	0
$\overrightarrow{mask_4}$	1	0	0	6	1
$\overrightarrow{mask_5}$	1	0	1	4	1
$\overrightarrow{mask_6}$	1	1	0	0	1
$\overrightarrow{mask_7}$	1	1	1	-1	0

Figure 8.1: A truth table finding all 2^3 linear combinations of $\vec{w} = [\ 3\ \ -6\ \ -2\ ]$ which satisfy $w_{act} \geq 0$. 3 is the bias constant and 0 is the threshold. Section 8.5.1 gives a linear algebra implementation for this brute force approach.

8.1 Brute Force

In this section, I give an example of a brute-force for-loop approach in algorithm 5. The $\overrightarrow{mask} \in (false, true)^{length(\overrightarrow{row})}$ for loop is illustrated in fig. 8.1 where each variable is defined and $w_{act} \geq 0$ is the conditional.

Preamble for algorithm 5 Let $n =$ the number of dimensions in $\vec{x}$, $SOP = false$ be the sum of products, and P be the current product. Let $\mathbb{L}$ be the ordinal set of layers in the neural network. Each $l \in \mathbb{L}$ is a double (W, B) where W is a matrix of weights with the ordinal of the previous layer's node found by the column number, and the current layer's node ordinal found by the row number. In other words, the weight $w_{i,j}$ in matrix W is yielded by the transition function $prev_j \rightarrow_w curr_i$. See fig. 8.2 for an illustration. B is the vector of bias weights for each node in the layer. Finally, "|" and ">>" are binary AIG operators for parallel composition and sequential composition, respectively.

Let the layers in NeuralNetwork be ordered as $\{1, 2, 3, \dots\}$. Since the input layer's (L_0) out-degree weights are remembered by L_1's in-degree weights, the algorithm will extract the input layer (L_0) from the first layer (L_1) in the neural network array and its nodes will be defined as $\{node_{0,1}, node_{0,2}, \dots, node_{0,n}\}$.

Algorithm 5: Converting a neural network to an AIG with brute force and for loops.

Input:

- NeuralNetwork, a two-dimensional array of weights following fig. 8.2.
- $\theta_t \in \mathcal{D}$ of $f(x)$, the threshold (θ_t) of the activation function $f(x)$ as a domain value ($\mathcal{D}$). See section 7.2 and section 7.3 for why $\theta_t \in \mathcal{D}$.

Output: NNAIG, the AIG representation of NeuralNetwork

```
NNAIG ← null
foreach layer_l ∈ NeuralNetwork do
    layerAIG ← null
    /* row⃗_i is a row vector that consists of all in-degree
       weights (except the bias weight) for the i^th node in
       layer l.                                              */
    foreach row⃗_i, bias ∈ layer do
        /* The SoP representing the i^th node of the l^th layer
           will be saved to node_{l,i}                       */
        node_{l,i} ← false
        /* Iterate over every possible bitmask              */
        forall mask⃗ ∈ (false, true)^{length(row⃗)} do
            if row⃗ · mask⃗ + bias ≥ θ_t then
                product ← true
                forall b_i ∈ mask⃗ do
                    /* node_{l-1,i} is the i^th node from the
                       previous layer.                       */
                    if b_i = 0 then
                        product ← product ∧ ¬node_{l-1,i}
                    else
                        product ← product ∧ node_{l-1,i}
                node_{l,i} ← node_{l,i} ∨ product
        layerAIG ← layerAIG|SoP
    NNAIG ← NNAIG >> layerAIG
return NNAIG
```

$$\begin{array}{cccc} prev_1 \to_w curr_1 & prev_2 \to_w curr_1 & \dots & prev_n \to_w curr_1 \\ prev_1 \to_w curr_2 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ prev_1 \to_w curr_m & \dots & \dots & prev_n \to_w curr_m \end{array}$$

(a) The layer l represented by a matrix of one-way transition functions $\to_w$ with operands of two nodes in adjacent layers where $prev_j$ is the prior layer.

$$\begin{array}{cccc} w_{1,1} & w_{1,2} & \dots & w_{1,n} \\ w_{2,1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ w_{m,1} & \dots & \dots & w_{m,n} \end{array}$$

(b) A simplified version of fig. 8.2a. For consistency with matrix notation, weights are labeled as $w_{curr_i,prev_j}$.

Figure 8.2

8.2 Binary Decision Tree ($\mathcal{B}$)

The classes of neural nodes are remembered by an enum and are true, false, and mixed. Mixed nodes are nodes that are not always true or false. $\mathcal{B}$ determines each neural node's class and proceeds to find the sum-of-products if the node is in the mixed class. If the node is true or false, $\mathcal{B}$ labels it as such in $\Theta(n)$ time. Otherwise, $\mathcal{B}$ has an average complexity of $O(2^{0.5n})$ and a worst-case complexity of $O(2^n)$ (see chapter 10).

During the node's parsing, $\mathcal{B}$ keeps track of which node in the previous layer was considered in this sum of products. When a node in the previous layer is never considered for the current layer, this node is labeled as "negligible."

As proven in section 7.1, ordering weights by importance is the same as ordering weights by the greatest absolute value—this is why $\mathcal{B}$ sorts the weight vector. Furthermore, recall the activation function is not computed due to section 7.2 and section 7.3. The definition of a threshold is given in lemma 7.2.2.

8.2.1 Order

To list the inputs from the greatest importance to the least importance I used the C++ standard library's (STL) std::sort which has a time complexity of $O(n\log(n))$. However, the ordinal value (position) of the weight was used to determine which node in the previous layer the weight came from. This ordinality is destroyed by sorting. To compensate, I created a simple struct called "ordinal_weight."

```
struct ordinal_weight{
  unsigned short const ordinal;
  float const weight;
  ordinal_weight(unsigned short const o, float const w)
      : ordinal(o), weight(w){}
  inline bool absGreaterThan(ordinal_weight const& rhs)
      const{
    return std::fabs(weight) > std::fabs(rhs.weight);
  }
};
```

Then STL's sort function allows for an easy sort.

```
std::sort(
  ordinalWeightsArr.begin(),
  ordinalWeightsArr.end(),
  [](ordinal_weight const& lhs, ordinal_weight const&
     rhs){
    return lhs.absGreaterThan(rhs);
  }
);
```

8.2.2 $\mathcal{B}$ defined

As described before, $\mathcal{B}$ is an optimized, binary decision tree which finds the sum-of-products (SoP) approximating a neural node. $\mathcal{B}$ illustrates $\mathcal{B}$ in pseudo-code.

To summarize, $\mathcal{B}$ keeps track of the current summation of weights (current), the minimum possible summation (min), and the maximum possible summation (max). If min is ever greater than or equal to zero, then any combination of future weights will yield true—therefore these weights are

"don't-cares" for this product. In the second case, if max is ever less than or equal to zero, then any combination of future weights will yield false. Finally, if neither case is true, the tree continues traversing. Note both cases include equals in their conditional, this is the greedy approach as foreshadowed in lemma 7.2.2.

Definitions Let π be the current path in $\mathcal{B}$ and binaryNodeArgs be a tuple

$$(currentSum, maxSum, minSum, node_{l,j}, i, product, SoP, prevNodesMatter)$$

where

- currentSum is the summation of weights given π.
- maxSum is the maximum possible summation of weights given π.
- minSum is the minimum possible summation of weights given π.
- $node_{l,j} = (w_0, o_0), (w_1, o_1), \ldots, (w_n, o_n)$ is an array of weight_ordinal pairs (see section 8.2.1) which consists of the in-degree weights for the j^{th} node of the l^{th} layer.
- (w_i, o_i) is the current weight_ordinal pair in $node_{l,j}$
- product is a truth array that will be updated and joined with the sum-of-products.
- SoP is the table of products.
- prevNodesMatter is a Boolean array which $\mathcal{B}$ uses to tell the network parser which nodes from the previous layer were used in $node_{l,j}$.

Procedure root($node_{l,j}, b, \theta_t$).

Input:

- $node_{l,j} = [(w_0, o_0), (w_1, o_1), \ldots, (w_n, o_n)]$, a vector of weight-ordinal pairs.
- b, a bias constant.
- θ_t, the θ-threshold (see section 7.3).
- negligible, a Boolean variable. If $node_{l,j}$ was not used for any node in the sequential layer, $\mathcal{B}$ is skipped entirely.
- prevNodesMatter, as described in section 8.2.2.

Output:

- SoP, a sum-of-products describing $node_{l,j}$.
- prevNodesMatter is manipulated and passed by reference.

if *negligible is true* **then**
  return
`/* Check if this node is always true or always false. */`
$b \leftarrow b - \theta_t$;
$currentSum \leftarrow b$;
$minSum \leftarrow b$;
$maxSum \leftarrow b$;
for $i \leftarrow 0 \ldots n$ **do**
  if $w_i \geq 0$ **then**
    $maxSum \leftarrow maxSum + w_i$
  else
    $minSum \leftarrow minSum + w_i$
if $maxSum < 0$ **then**
  return false
else if $minSum \geq 0$ **then**
  return true
`/* Sorted by weight value. */`
$node_{l,j} \leftarrow sort(node_{l,j})$;
`/* Initialize values */`
$product \leftarrow true$;
$SoP \leftarrow false$;
`/* Begin the algorithm */`
$binaryNodeArgs \leftarrow$
  $(currentSum, maxSum, minSum, node_{l,j}, 0, product, SoP, prevNodesMatter)$;

if $w_0 \geq 0$ **then**
  pos_true($binaryNodeArgs$);
  pos_false($binaryNodeArgs$);
else
  neg_true($binaryNodeArgs$);
  neg_false($binaryNodeArgs$);
return SoP

Procedure atomize(l, o) –a name mangling function that labels each given node from the neural network.

Input:

- l, the layer number.
- o, the node's ordinal value.
- prevNodesMatter, as defined in section 8.2.2.

Output: A unique name for $node_{l-1,o}$ to be used in labeling AIG nodes.

prevNodesMatter[o] $\leftarrow$ true;

```
/* The ordinal value, o, is referring to the previous
   layer; hence, the name-wrangling does ‘‘l-1".       */
```

return *"L" + string(l-1) + "N" + string(o)*

Procedure directions($binaryNodeArgs$)

Input: binaryNodeArgs

$i \leftarrow i + 1$;

if $w_i \geq 0$ **then**

  posTrue($binaryNodeArgs$);

  posFalse($binaryNodeArgs$);

else

  negTrue($binaryNodeArgs$);

  negFalse($binaryNodeArgs$);

return

Procedure posTrue(binaryNodeArgs). The true case of a positive weight.

Input: binaryNodeArgs
$product \leftarrow product \wedge atomize(o_i)$;
$current \leftarrow current + w_i$;
$minSum \leftarrow minSum + w_i$;
if $minSum \geq 0$ **then**
  $SoP \leftarrow SoP \vee product$
else
  directions(binaryNodeArgs)
return

Procedure posFalse(binaryNodeArgs). The false case of a positive weight.

Input: binaryNodeArgs
$product \leftarrow product \wedge \neg atomize(o_i)$;
$max \leftarrow max - w_i$;
if $max \geq 0$ **then**
  directions(binaryNodeArgs)
return

Procedure negTrue(binaryNodeArgs). The true case of a negative weight.

Input: binaryNodeArgs
$product \leftarrow product \wedge atomize(o_i)$;
$current \leftarrow current + w_i$;
$max \leftarrow max + w_i$;
if $max \geq 0$ **then**
  directions(binaryNodeArgs)
return

Procedure negFalse(binaryNodeArgs). The false case of a negative weight.

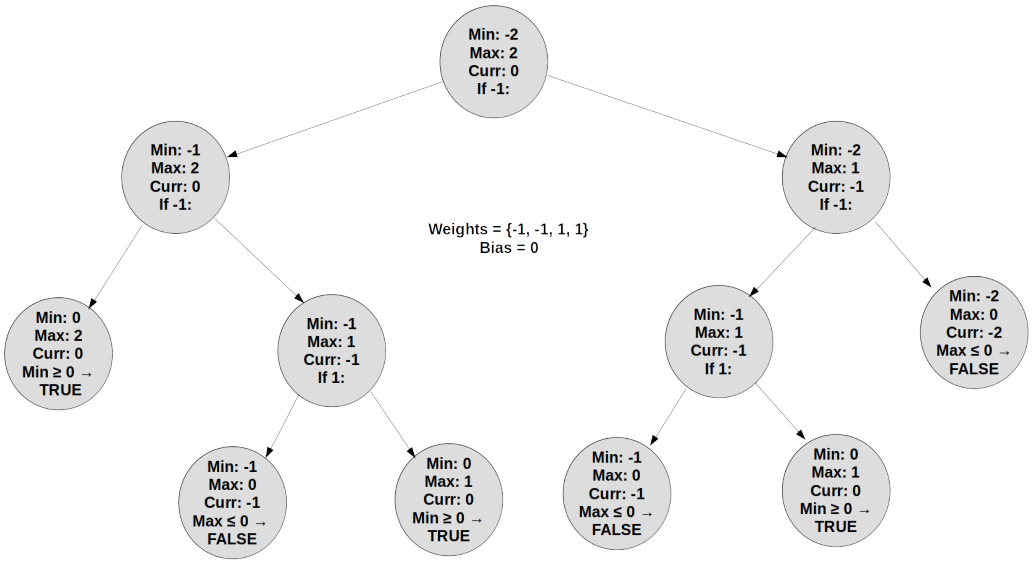
Input: binaryNodeArgs
$product \leftarrow product \wedge \neg atomize(o_i)$;
$min \leftarrow min - w_i$;
if $min \geq 0$ **then**
  $SoP \leftarrow SoP \vee product$
else
  directions(binaryNodeArgs)
return

8.2.3 Complexity

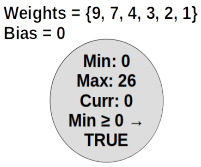
While the sum-of-products of a node in the worst case is exponential ($O(2^n)$) due to the Cook-Levin theorem [7], $\mathcal{B}$ is able to have an average complexity of $O(2^{0.5n})$ (see chapter 10) and a best-case complexity of $\Omega(n)$.

Best-case Consider the case where all weights and the bias are positive and the threshold is zero. $\mathcal{B}$ will iterate over each weight (costs $\Theta(n)$) to find the maximum and minimum sums possible. Since every weight and the bias are positive, the minimum sum will be equal to the bias which is greater than zero. $\mathcal{B}$ ends here and notes that this neural node is always true. The $O(n \times \log_2(n))$ sort nor the tree traversal are needed. Therefore, $\mathcal{B}$'s best case is $\Theta(n)$. See fig. 8.3b for an example.

Worst-case $\mathcal{B}$'s worst case is $O(2^n)$ and occurs when all of the weights have equal magnitudes and are as evenly distributed as possible between the positives and negatives bins (a weight vector with an odd size cannot have equal distribution between the positives and negatives). Figure 8.3a illustrates this case. However, section 10.2.4 shows that $\mathcal{B}$'s worst case is still approximately twice as fast as the brute-force approach.



(a) The worst-case scenario is $\Theta(2^n)$, and is quicker than the brute force method ($\Theta(2^n n)$) by a factor of n (see chapter 10).



(b) The best-case scenario is $\Theta(n)$. Technically, the tree's best complexity is $\Theta(1)$, but $\mathcal{B}$ must parse the weights in $\Theta(n)$ time before the tree starts. The best-case is quicker than the brute-force approach by a factor of 2^n.

Figure 8.3: Visualization of the possible complexity extremes.

8.3 Traversing the Neural Network

The neural network's reversed traversal allows the algorithm to decide if a neural node must be predicted by $\mathcal{B}$ as defined in $\mathcal{B}$ or if the neural node is negligible and subsequently skipped. This is tracked by the prevNodesMatter argument in $\mathcal{B}$. Algorithm 6 describes a problematic forward traversal which calculates $\mathcal{B}$ for every node while algorithm 7 is the optimized reverse traversal.

Forward traversal Algorithm 6 can be naïvely parallelized by node or by layer. However, algorithm 6 can also produce islands or dead ends in the final AIG. For example, if no sum-of-products produced by layer 2 include some node, $node_{1,i}$, in layer 1, then the AIG's will have a dead-end representing $node_{1,i}$. An island may occur if a node is determined to be constant (always true or always false) and is negligible (not used in any SoP for the subsequent layer). Algorithm 7 modifies algorithm 6 by omitting negligible nodes from computation and the AIG output.

Algorithm 6: Forward Traversal

Input: $NN = [l_1, l_2, \ldots, l_m]$. Each $l_i \in NN$ is a vector of nodes $[node_0, node_1, \ldots, node_n]$. Each $node_j \in l_i$ is a vector of weight-ordinal pairs (w_k, o_k) where o_k is the placement of w_k in the original input vector for $node_j$. The original placement signifies the associated node from the previous layer. The function *root* is as defined in $\mathcal{B}$ and begins $\mathcal{B}$.

Output: AIG approximation of the neural network.

for $i = 1 \ldots m$ **do**
  forall $node_j \in l_i$ **do**
    $SoP_{i,j} \leftarrow root(node_{i,j}, bias, \theta_t)$;
    Label the output of $SoP_{i,j}$ as $atomize(i, o_j)$.;
  $ParallelAIG_i \leftarrow SoP_{i,0} | SoP_{i,1} | \ldots | SoP_{i,n}$;
$CompleteAIG \leftarrow ParallelAIG_1 >> ParallelAIG_2 >> \cdots >> ParallelAIG_m$;
return $CompleteAIG$

Reverse traversal Algorithm 7 reduces the computation needed by omitting the negligible neural nodes which also prevents dead ends and islands.

Algorithm 7: $\mathcal{N}$: Reverse traversal

Input: $NN = [l_1, l_2, \ldots, l_m]$ as described in algorithm 6.
Output: AIG approximation of the neural network.
Let $length(l_i)$ return the number of nodes in l_i.;
$DoCurrentInputsMatter \leftarrow [true_0, true_1, \ldots, true_{length(l_m)}]$;
for $i = m, m-1, \ldots, 1$ **do**
  $DoPrevInputsMatter \leftarrow [false_0, false_1, \ldots, false_{length(l_{i-1})}]$;
  forall $node_{i,j} \in l_i$ **do**
    $SoP_{i,j} \leftarrow$
     $root(node_{i,j}, bias, \theta_t, DoCurrentInputsMatter[j], DoPrevInputsMatter)$;
  $DoCurrentInputsMatter \leftarrow DoPrevInputsMatter$;
  $ParallelAIG_i \leftarrow SoP_{i,0}|SoP_{i,1}|\ldots|SoP_{i,n}$;
$CompleteAIG \leftarrow ParallelAIG_1 >> ParallelAIG_2 >> \cdots >>$
 $ParallelAIG_m$;
return $CompleteAIG$

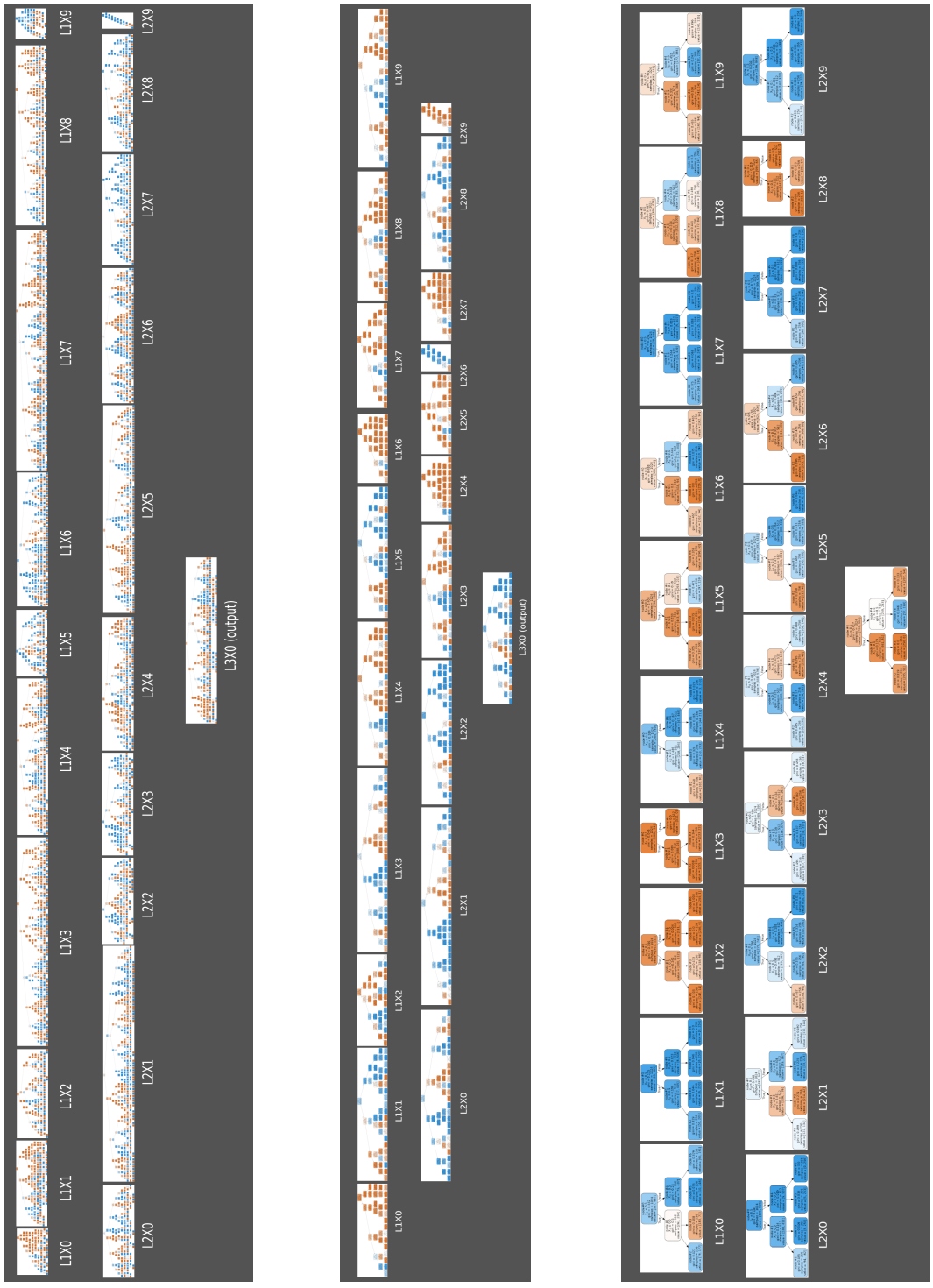
8.4 Additional Ideas

8.4.1 Maximum Tree Depth

To reduce the worst case complexity, a maximum depth (d) can be enforced. The worst-case complexity will become $O(2^d)$ and the average case will be $O(2^{0.5n})$ (recall n is the size of the weight array) if $d > 0.5n$ else the average complexity will be $O(2^d)$. However, the maximum depth will give some loss in accuracy for instances which $\mathcal{B}$ would normally surpass d.

Maximum tree depth has been implemented in the source code for $\mathcal{B}$ ($\mathcal{B}$).

Recall the XORNN from chapter 6 was only composed of three layers. If we change the first two layers from two to ten inputs, the average amount of nodes in the decision trees increases exponentially. In fig. 8.4a, no maximum depth is enforced. However, the optimal solution's depth cannot exceed the size of the previous layer (10 in this case); the proof is in chapter 7 and $\mathcal{B}$ enforces this concept.



(a) No max depth given. (b) Max depth = 5 (c) Max depth = 2

Figure 8.4: Three decision tree networks each representing a 10-10-10-1 (inputs, hidden layer 1, hidden layer 2, output) layered neural network. The input layer is inserted into the first layer of decision trees, then the network proceeds through the layers in the same fashion a neural network predicts input–the only difference being $\mathcal{B}$ is now the activation function.

8.4.2 Subtrees

Since $\theta = \vec{x} \cdot \vec{w}^T$ is a linear combination, coefficients (weights) with similar ratios and positivity yield similar activations. This causes some subtrees in $\mathcal{B}$ to be identical. Future work may determine how to recognize these repeated structures.

8.5 Alternatives

8.5.1 Linear Algebra

Alternatively, linear algebra could be used to obtain the sum-of-products for a node. Below is the Python source code that can be used to compute an entire layer (all input weight sizes are the same since the network is fully connected).

```
import numpy as np
from itertools import product
sizeOfPrevLayer = 10
sizeOfThisLayer = 1
#columns are the weights from the previous layer to the
    ith (row) node of the current layer
weights = np.random.uniform(-10, 10, (sizeOfPrevLayer,
   sizeOfThisLayer))
inputs = np.asarray(list(product([0,1], repeat=
   sizeOfPrevLayer))) #2^n possibilities
outputs = np.matmul(inputs, weights) >= 0
#Construct truth tables by concatenating inputs and
   outputs. A layer of nodes expressed as a layer of
   sum-of-products can be derived from this.
...
```

The iterative algorithm of matrix multiplication places this algorithm at $\Theta(2^n \times n \times sizeOfLayer)$. However, Raz that proved the lower bound for matrix multiplications of two **square** matrices is $\Omega(n^2 \log_2 n)$ [50]. Currently, there exists no algorithm with this low complexity—Alman and Williams published the current best complexity of $O(n^{2.37286})$ in October 2020 [51].

Theorem 8.5.1. *Let s be the size of the current layer and n be the size of the weight array (equal the size of the previous layer). Then, the matrix approach has a lower bound of $\Omega(2^n n \log_2 n)$ if $n \leq s$ and $\Omega(2^n s \log_2 s)$ if $n \geq s$.*

Proof. If $n \leq s$, then let's assume the matrix product $M = 2^n \times n \cdot n \times s$ is instead $C = 2^n \times n \cdot n \times n$ which is a quicker computation than the original.

Split the left-hand side of C ($2^n \times n$) into $\frac{2^n}{n}$ submatrices each with size $n \times n$ and having no overlap. Then the product of each submatrix with the reduced RHS ($n \times n$) has a lower bound of $\Omega(n^2 \log_2 n)$ by Raz's theorem. Since there are $\frac{2^n}{n}$ submatrices, C is therefore in

$$\Omega(\frac{2^n}{n} n^2 \log_2 n) = \Omega(2^n n \log_2 n)$$

Since M is more computationally expensive than C, M is also in $\Omega(2^n n \log_2 n)$.

Alternatively, $n > s$. Like the prior proof, I reduce M to a less expensive C matrix. Let $C = 2^n \times s \cdot s \times s$.

Following the same logic as before, split $2^n \times s$ into $\frac{2^n}{s}$ submatrices. Then, C is in

$$\Omega(2^n n \log_2 n)$$

Since M is more computationally expensive than C, M is also in $\Omega(2^n n \log_2 n)$. □

Corollary 8.5.1.1. *With a worst-case complexity of $O(2^n)$ per node, $\mathcal{B}$ is still more computationally efficient when computing an entire layer $O(s \times 2^n)$ than the theoretical optimal algorithm for the linear algebra approach of $\Omega(2^n n \log_2 n)$.*

8.5.2 Another Approach

First and foremost, this algorithm has a worse memory complexity and average time complexity than $\mathcal{B}$. This approach attempted to exploit tables to reduce the worst case of $O(2^n)$ to $O(2^{0.5n} n)$ which occurs when the weights are approximately equal in absolute value and distribution of positive and negative weights. Unfortunately, this algorithm's time complexity is $\Theta(2^n n)$ and the memory complexity is in $\Omega(2^{\frac{n}{2}} n)$ and $O(2^n n)$. In comparison, $\mathcal{B}$ has a time and space complexity of $\Omega(n)$ to $O(2^n)$ with an average of $O(2^{0.5n})$ (see the complexity of nodes in section 10.1).

This algorithm was appealing due to the potential $\Omega(2^{\frac{n}{2}})$ memory-bound (see table 8.1) and the binary search shown in step 4. This algorithm's best-case occurs exactly in the worst cases of $\mathcal{B}$ and was intended to only be used in those cases.

Step 0: Begin with a set of weights, W.

Step 1: Divide W into sorted negative and positive bins; negligible weights (0 plus or minus the machine epsilon or greater if the user desires) can be omitted. This step is done in $\Theta(n)$.

Step 2: For each possible bitmask in both bins, calculate and remember the respective subsum (can be done by matrix multiplication section 8.5.1). This step is in $\Omega(2^{0.5n} n \log_2 n)$ if the negative and positive bins are equal in size and $\Omega(2^n n \log_2 n)$ on the other extreme.

Step 3: Sort both bins. If step 3a can be done, this step will be a constant $O(1)$, else it will take $\Omega(2^{n/2} \times \log_2(2^{n/2}) = \Omega(2^{n/2} n)$ and $O(2^n n)$.

Step 3a: Depending on the data set, the sort may be done inherently in step 2. For both bins, if the sum of the two smallest (in absolute value) is greater than or equal to the greatest absolute value of that same bin, a bitmask pattern as done in table 8.1 can be done to produce a completely sorted array. In NEG from table 8.1, the two smallest absolute values are {-1,-0.9} and the greatest absolute value is -1.1. Now, $|-1 + -0.9| \geq |-1.1|$ so this set of weights can be sorted inherently. Omitting the negligible weights in step 1 allows more sets to be inherently sorted given this bitmask pattern. Furthermore, this algorithm was planned to only occur when the weights have approximately equal absolute values. These cases often allow for step 3a.

Algorithm 8: Step 4

Input:

- POS, a bin of subsums for all possible positive weights with the respective bitmask table.
- NEG, a bin of subsums for all possible negative weights with the respective bitmask table.

See table 8.1 for an example.

Output: NodeSoP. A sum-of-products explaining the current node.

if $sizeof(NEG) > sizeof(POS)$ **then**
  $bin_s \leftarrow POS$
  $bin_l \leftarrow NEG$
else
  $bin_s \leftarrow NEG$
  $bin_l \leftarrow POS$
$NodeSop \leftarrow False$
foreach $(Bitmask_i, subsum_i) \in bin_s$ **do**
  $Bitmask_n \leftarrow BinarySearch(bin_l, subsum_i)$ `/* BinarySearch returns the bitmask that corresponds to the subsum which is the closest absolute value in` bin_l `that is smaller than` $subsum_i$ `*/`
  if $Bitmask_n$ *is null* **then**
    continue
  foreach $Bitmask_j \in Bitmask_1 \dots Bitmask_n$ **do**
    $product \leftarrow bin_s(Bitmask_i) \cup bin_l(Bitmask_j)$
    $NodeSop \leftarrow NodeSop \vee product$
return *NodeSop*

Analysis of algorithm 8 Let s be the size of bin_s and l be the size of bin_l where $0 \leq s \leq 2^{\frac{n}{2}}$, $2^n \geq l \geq 2^{\frac{n}{2}}$, and $s \times l = 2^n$. Finding each $Bitmask_n$ in step 4 is in $O(s \log_2(l))$ which has the worst case when $s = l$. In terms of n, this makes the complexity $O(2^{\frac{n}{2}} \log_2(2^{\frac{n}{2}})) = O(2^{\frac{n}{2}} n)$ to find each $Bitmask_n$. Unfortunately, adding each lower bitmask in the inner loop makes the algorithm $O(sl) = O(2^n)$.

Algorithm 8 example: In table 8.1, $subsum_s$'s (POS bin) first subsum s_1 is zero which is paired with Bitmask1. Searching the NEG bin for the absolute value closest to s_1 and still less than s_1 yields Bitmask1, therefore Bitmask1 of POS unioned with Bitmask1 of NEG yields a total summation greater than or equal to zero–therefore this combination is a product in the sum-of-products representing the node (SoP_{node}). $subsum_s$'s second subsum s_2 is 1 (Bitmask2) which finds the closest, smaller absolute value in NEG bin at Bitmask2 so Bitmask2 of NEG unioned with Bitmask2 of POS is also a product in SoP_{node}. Additionally, each BitmaskN in NEG below Bitmask2 unioned with Bitmask2 of POS is also in SoP_{node}—here that only includes Bitmask1 of NEG. So far, $SoP_{node} = 000000 + 001001 + 000001$ where the 000000 comes from the iteration of Bitmask1 in POS, and $001001 + 000001$ comes from Bitmask2 in NEG.

To assert correctness (not needed in the algorithm—only for human understanding), add the respective subsums for each product in the SoP_{node} obtained so far:

$$\begin{aligned} 000000 &\rightarrow NEG(000) + POS(000) = 0 + 0 = 0 \geq 0 \\ 001001 &\rightarrow NEG(001) + POS(001) = -0.9 + 1 = 0.1 \geq 0 \\ 000001 &\rightarrow NEG(000) + POS(001) = 0 + 1 = 1 \geq 0 \end{aligned}$$

Table 8.1: An example of step two in section 8.5.2 with the set of weights being {1.2,1.1,1,-1.1,-1,-0.9} and the bias constant B = 0.

	NEG			Negative Subsums	POS			Positive Subsums
Weights	-1.1	-1	-0.9	BitmaskN·NEG+B	1.2	1.1	1	BitmaskN·POS+B
Bitmask1	0	0	0	0	0	0	0	0
Bitmask2	0	0	1	-0.9	0	0	1	1
Bitmask3	0	1	0	-1	0	1	0	1.1
Bitmask4	1	0	0	-1.1	1	0	0	1.2
Bitmask5	0	1	1	-1.9	0	1	1	2.1
Bitmask6	1	0	1	-2	1	0	1	2.2
Bitmask7	1	1	0	-2.1	1	1	0	2.3
Bitmask8	1	1	1	-3	1	1	1	3.3

Conclusion This algorithm was promising because of the binary search in algorithm 8 and only computing $2^{0.5n}$ subsums in the best case. However,

even if we assume the best matrix algorithm possible for the best case in step 2 ($\Omega(2^{0.5n} n \log_2 n)$) and a constant sort in step 3, step 4 will still make this algorithm's optimal scenario $O(2^n)$. The memory and time complexities of this algorithm are worse than $\mathcal{B}$.

Chapter 9

Training with don't-care variables

9.1 Red is stop

First, let's begin with a simple Boolean function.

RYG RYG is a simple function with two don't care variables:

$$RYG(R, Y, G) = \neg R$$

Table 9.1 shows how a neural network can discover this relationship (simplified with $\mathcal{N}$). See fig. 9.1a for the AIG representation of this table. Next, ABC further simplifies the function to fig. 9.1b.

Table 9.1: Every row is a product in the sum-of-products for each of the neural nodes trained on the RYG function in a $3 \times 2 \times 1$ (excluding biases) neural network.

L2X1

r	y	g
0	-	-

L2X2

r	y	g
0	-	0

L3X1

L2X1	L2X2
1	-
0	1

Theorem 9.1.1. *$\mathcal{N}$'s approximation of the neural network (shown in table 9.1) is equivalent to RYG.*

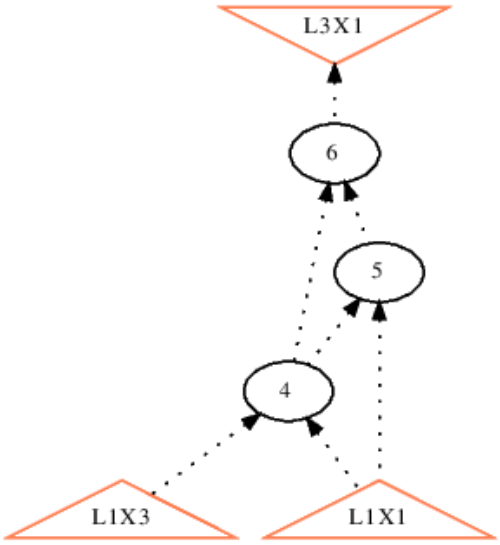
Proof. First, convert table 9.1 to Boolean functions.

$$L2X1 = \neg r \tag{9.1}$$
$$L2X2 = \neg r \neg g \tag{9.2}$$
$$L3X1 = L2X1 \vee \neg L2X1 \wedge L2X2 \tag{9.3}$$

Using substitution then simplification,

$$L3X1 = \neg r \vee \neg(\neg r) \wedge \neg rg \tag{9.4}$$
$$= \neg r \vee r \neg rg \tag{9.5}$$
$$= \neg r \vee 0 \tag{9.6}$$
$$= \neg r \tag{9.7}$$

□



(a) Table 9.1 translated to an AIG without neural-trans-layer simplification.

(b) Figure 9.1a simplified to the optimal level of comprehension.

Figure 9.1: AIGs derived from table 9.1. L1X1 is red and L1X3 is green. $\mathcal{N}$ was able to optimize away L1X2 (yellow). This serves as a secondary proof that table 9.1 is equivalent to RYG.

Discussion Interestingly, the neural network allowed for an impossible case of $\neg r \wedge r$ in one of the products. This opens the door for future work in $\mathcal{B}$. Perhaps the scenario is possible before the binarization and this quasi impossibility could be used to achieve a better understanding of the neural network. Alternatively, this scenario is impossible and future work could help $\mathcal{B}$ skip these impossible scenarios. Another instance of $\mathcal{B}$ including unneeded values is seen in section 9.2.3.

9.2 Iris Data set

9.2.1 Real to Binary Inputs

Before real-valued input is used, it must be cast to a Boolean value (binarized). Algorithm 9 defines how binarization is done in this chapter.

Algorithm 9: Binarization algorithm.

Input:

- $\hat{x}$, some threshold.
- $I \in \mathbb{R}^n$, the set of inputs in the real domain.

Output: $I_{\text{binarized}} \in \mathbb{B}^n$.
forall $x \in I$ **do**

$$x \leftarrow \begin{cases} 0 & \text{if } x < \hat{x} \\ 1 & \text{otherwise.} \end{cases}$$

return I

Binarization thresholds Preliminary investigations tested four methods to obtain a threshold: median, mean, K-means with one centroid (took the centroid's value), and K-means with two centroids (took the average of the two centroid values). The mean threshold appeared to give the most accurate results. Future work can further analyze this threshold.

9.2.2 Illustrating Limitations of Algorithm 9

Two limitations are identified in this section: binarized data sets that produce all possibilities and binarized data sets which do not accurately represent the original data.

Higgs

Consider the Higgs data set [52]. This set is split into high-level values (produced by functions) and low-level values (basic data used to compute the high-level features). A goal with this data set is to find better functions to predict if the given data is noise or an important signal. To do so, a neural network can be trained on the data then analyzed.

Neural Network Results After applying the binarization algorithm in algorithm 9 with the mean threshold, I was able to get 67% accuracy and 0.67 AUC by implementing a 28X24X24X1 neural network with the swish function activation (the majority class is 53%). Using real numbers and a far more intricate deep neural network, Baldi, Sadowski, and Whiteson managed to get 88% accuracy and 88 AUC in [52]. However, the AIG constructed from my 28X24X24X1 neural network (with binarized input) generated a file size too large (greater than 4GB) for ABC. So I was unable to use ABC's simplification tools.

The loss in accuracy by binarizing the Higgs data set can be better understood by looking at the binarized Iris data set in section 9.2.2.

A simpler neural network Since the 28X24X24X1 elicited an AIG too large for ABC to simplify, I tested a simpler neural network with fewer inputs.

I used Pearson correlation on the low-level values to approximate the five most promising inputs for a simple 5X4X4X1 neural network. Upon testing, I discovered the neural network performed poorly. After analyzing the data set, I found every one of the 2^6 possible Boolean combinations was present. See fig. 9.2 for an example of every Boolean combination given two inputs.

AIGs from uncertain neural networks Furthermore, the resulting AIG's approximation of the simpler neural network had varying accuracy (about 50% to 90%). This is likely due to the activation outputs being around the binary threshold caused by the trained neural network's uncertainty. See

a	b	f(x)
0	0	0
0	0	1
0	1	0
1	0	0
0	1	1
1	0	1
1	1	0
1	1	1

Figure 9.2: A neural network cannot learn this function when reduced to this set, yet when trained on the original set the neural network biased towards the most instances of input/output pairs.

fig. 9.9 for more research into neural network certainty and resulting AIG approximations.

Binarization Limits

Some features cannot be binarized without a sufficient loss of information.

For example, notice how Virginica's sepal width (orange lines) in fig. 9.3 tends to be less than Setosa's but greater than Versicolor's sepal widths. Assume we wanted to create a binary class of "yes Virginica" or "not Virginica" and binarize the respective inputs. With algorithm 9, we are allowed one threshold ($\hat{x}$) for each input. Where should the $\hat{x}$ for sepal width be? If $\hat{x} = 3$ then both 0 and 1 have approximately a $\frac{1}{3}$ chance to represent Virginica's sepal width—this makes the sepal width input useless. A better solution is to place $\hat{x}$ between Virginica and Versicolor ($\hat{x} \approx 2.8$), then 1 will have about 50% chance of being Virginica and 0 will be not Virginica.

However, this problem is better solved if we gave sepal width more thresholds and more bits as done in algorithm 10. Future work can expand on converting a real-valued input to multiple binary inputs (even greater than two) which can preserve essential information. Brudermueller et al. converted real-valued numbers to multiple binary inputs, but they did not minimize the number of bits needed per input [3].

Since algorithm 9 cast the real-valued inputs into binary with a single pivot, the optimal data set for this algorithm consists of important inputs (inputs with the most weight in deciding classes) that are linearly separable—

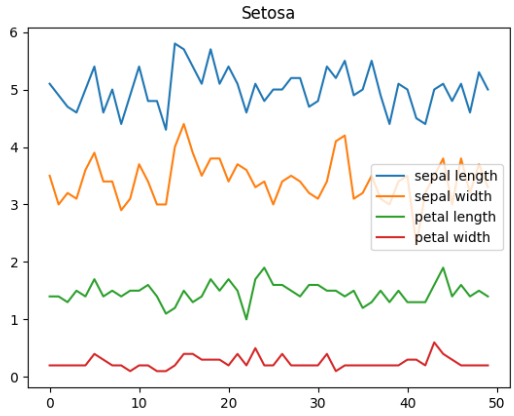
Algorithm 10: Example of casting a real-valued input into multiple binary inputs.

Input:

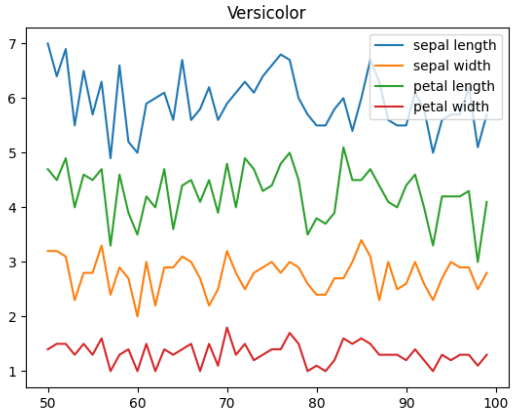
- $I_s \in \mathcal{R}^n$, the set of all sepal width inputs.
- $\hat{x}_1 = 2.8$, the first threshold for sepal width.
- $\hat{x}_2 = 3.2$, the second threshold for sepal width.

Output: $I_{s:2-bit}$, a set representing each element in I_s as a pair of bits.

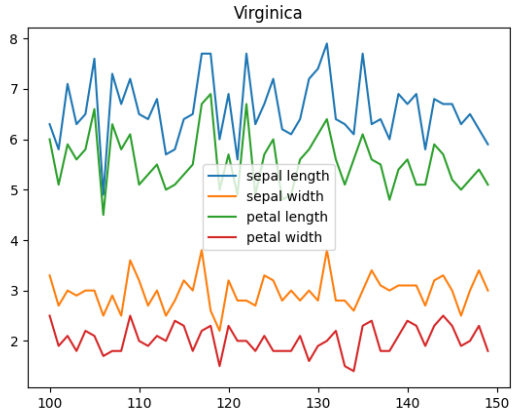
$I_{s:2-bit} \leftarrow \emptyset$
forall $x \in I_s$ **do**
  if $x < \hat{x}_1$ **then**
    $I_{s:2-bit} \leftarrow I_{s:2-bit} \cup \{00\}$
  else if $x < \hat{x}_2$ **then**
    $I_{s:2-bit} \leftarrow I_{s:2-bit} \cup \{01\}$
  else
    $I_{s:2-bit} \leftarrow I_{s:2-bit} \cup \{10\}$
return $I_{s:2-bit}$



(a)



(b)



(c)

Figure 9.3: Line plots of the features for each class in the renown Iris data set.

such as two clusters. Inputs which should not be determined by a single threshold will lose useful information (by definition) and should be discarded. This is why the future work of casting real-values to multiple bits is vital. Linearly separable relationships are the reason $\mathcal{B}$ can binarize activation functions in section 7.2.

Alternatively, one may train two neural networks–one that predicts Versicolor or Setosa, and the other predicts Versicolor or Virginica. To begin this second approach, copy the Iris data set into a second data set, then erase all Virginica instances from the data set for the first neural network, and all Setosa rows from the other. Now, sepal width is linearly separable in both data sets. The first neural network is explored in section 9.2.3.

9.2.3 Versicolor or Setosa

Generalization Captured I ran a 4X2X1 (bias nodes excluded) neural network several times on the Versicolor/Setosa binarized data set with the only difference being the initial randomized weights. Over 20 tests, the constructed AIG and the original neural network predicted class 0 for inputs with "–00" and class 1 for inputs like "–11". Six of those eight possibilities were explicitly trained in the neural network (fig. 9.4).

sepal length	sepal width	petal length	petal width	class
0	1	0	0	0
0	0	0	0	0
1	1	0	0	0
1	1	1	1	1
1	0	1	1	1
0	0	1	1	1

Figure 9.4: All unique instances from the 100 rows left from excluding the Virginica class/feature combinations. Each input is binarized by the mean threshold (section 9.2.1). Class 0 is Setosa and class 1 is Versicolor.

Theorem 9.2.1. *For the AIG in fig. 9.5, inputs that satisfy (-, -, 0, 0) must be class 0.*

Proof. Let (n3, n4) = (0, 0). Then

$$Node_8 = AND(\neg n3, \neg n4) = 1 \tag{9.8}$$

The network contains 4 logic nodes and 0 latches.

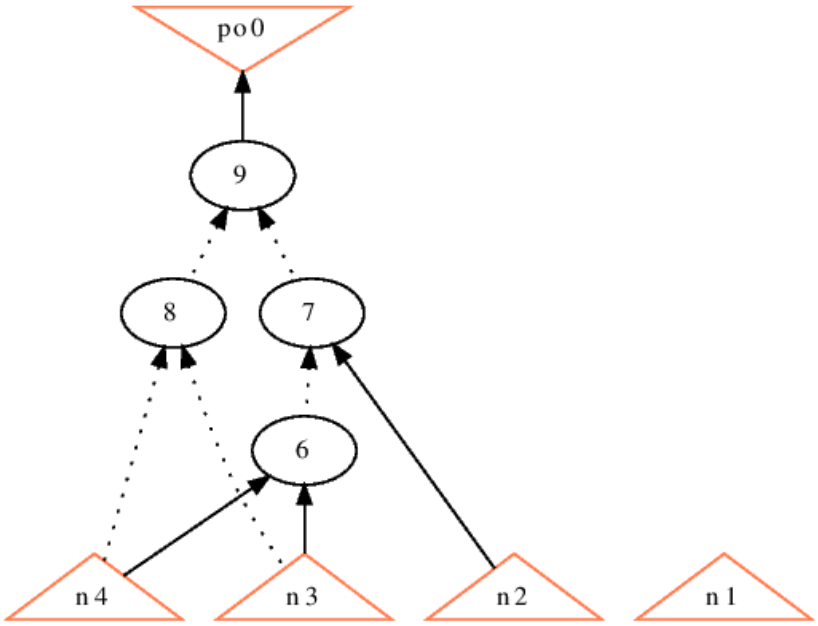


Figure 9.5: The AIG constructed by the weights given in fig. 9.6; (n1, n2, n3, n4) = (sepal length, sepal width, petal length, petal width). From this, we can illustrate the neural network places heavy importance on homogeneous petal length/width pairs (see theorem 9.2.1 and theorem 9.2.2).

In the binary AND operation, if any input is zero, then the result is zero.

$$Node_9 = AND(\neg Node_8, -) = 0 \tag{9.9}$$

The edge $Node_9 \rightarrow po0$ does not negate so

$$Node_9 = po0 \tag{9.10}$$

□

Theorem 9.2.2. *For the AIG in fig. 9.5, inputs that satisfy (-, -, 1, 1) must be class 1.*

Proof. Let (n3, n4) = (1, 1). Then

$$Node_8 = AND(\neg n3, \neg n4) = 0 \tag{9.11}$$

$$Node_6 = AND(n3, n4) = 1 \tag{9.12}$$

$$Node_7 = AND(\neg Node_6, -) = 0 \tag{9.13}$$

$$Node_9 = AND(\neg Node_8, \neg Node_7) = po0 = 1 \tag{9.14}$$

□

Noise While the AIG (fig. 9.5) accurately approximated the neural network for homogeneous petal length and petal width pairs, it was incorrect for inputs that the neural network was uncertain of (heterogeneous petal length and petal width values such as (-,-,0,1) or (-,-,1,0)). Given the data set, these heterogeneous pairs seem impossible or highly unlikely. Future work can consider the input space with $\mathcal{N}$ and omit the neural network relationships specifically for these unlikely inputs for a simpler AIG output.

Across several tests, the resulting AIG from the 4X2X1 neural network all articulated the homogeneous sepal length/width pairs with the predicted class correctly. The small neural network clearly gave this relationship priority. The other, more subtle, relationships (predicting classes for heterogeneous sepal length/width pairs) rose inconsistently and were often not helpful. Furthermore, several subtle relationships may not be worth considering if the goal is human comprehension (see fig. 9.8).

Despite that the 4X2X1 neural network was uncertain for most of the heterogeneous petal length/width inputs, section 9.3 shows that larger neural networks have room to predict heterogeneous pairs with greater certainty (estimation not around 50%) and elicit AIGs that correctly approximate the neural network (with about 95% accuracy). Section 9.3 is the first step to finding accurate (and relatively concise) AIGs for complicated neural networks by using ensembles.

```
Hidden node 1: -6.009057,2.0612597,-2.9449704,5.974104,6.9705567;
Hidden node 2: -5.9144406,1.9556314,-2.8650305,6.964409,5.840615
Output node: -12.945059,13.440295,13.615166
```

Figure 9.6: An example of a 4X2X1 neural network that is certain for homogeneous sepal length/width values and uncertain for other inputs. Each of the hidden nodes's in-degree weights are ordered as bias, sepal length, sepal width, petal length, and petal width. The output node in-degree weights are ordered as bias, hidden node 1, hidden node 2.

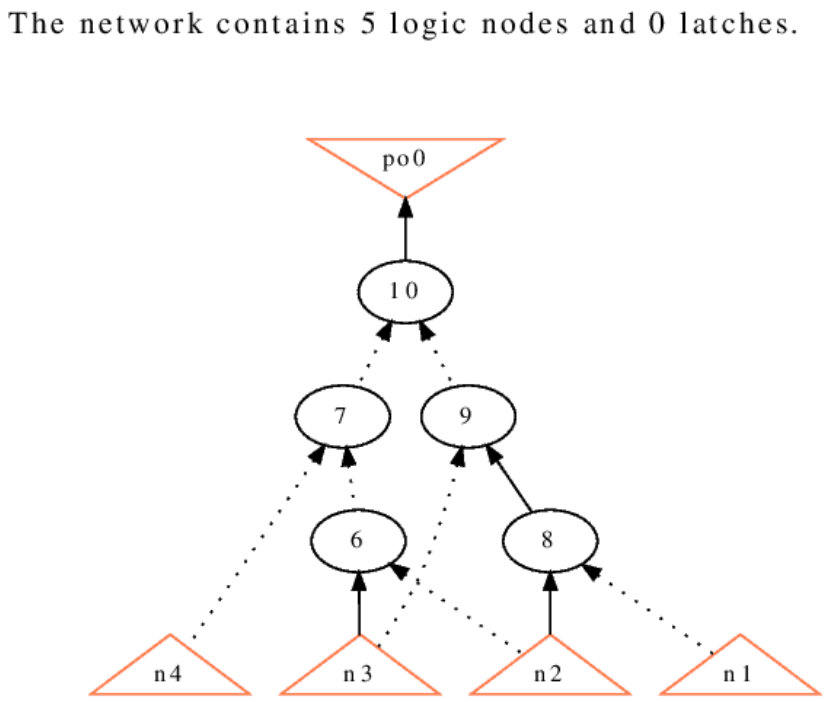


Figure 9.7: An AIG constructed from a separate neural network trained on the same data as the neural network that elicited fig. 9.5. We see that any input of homogeneous (n3, n4) pair is 100% accurate.

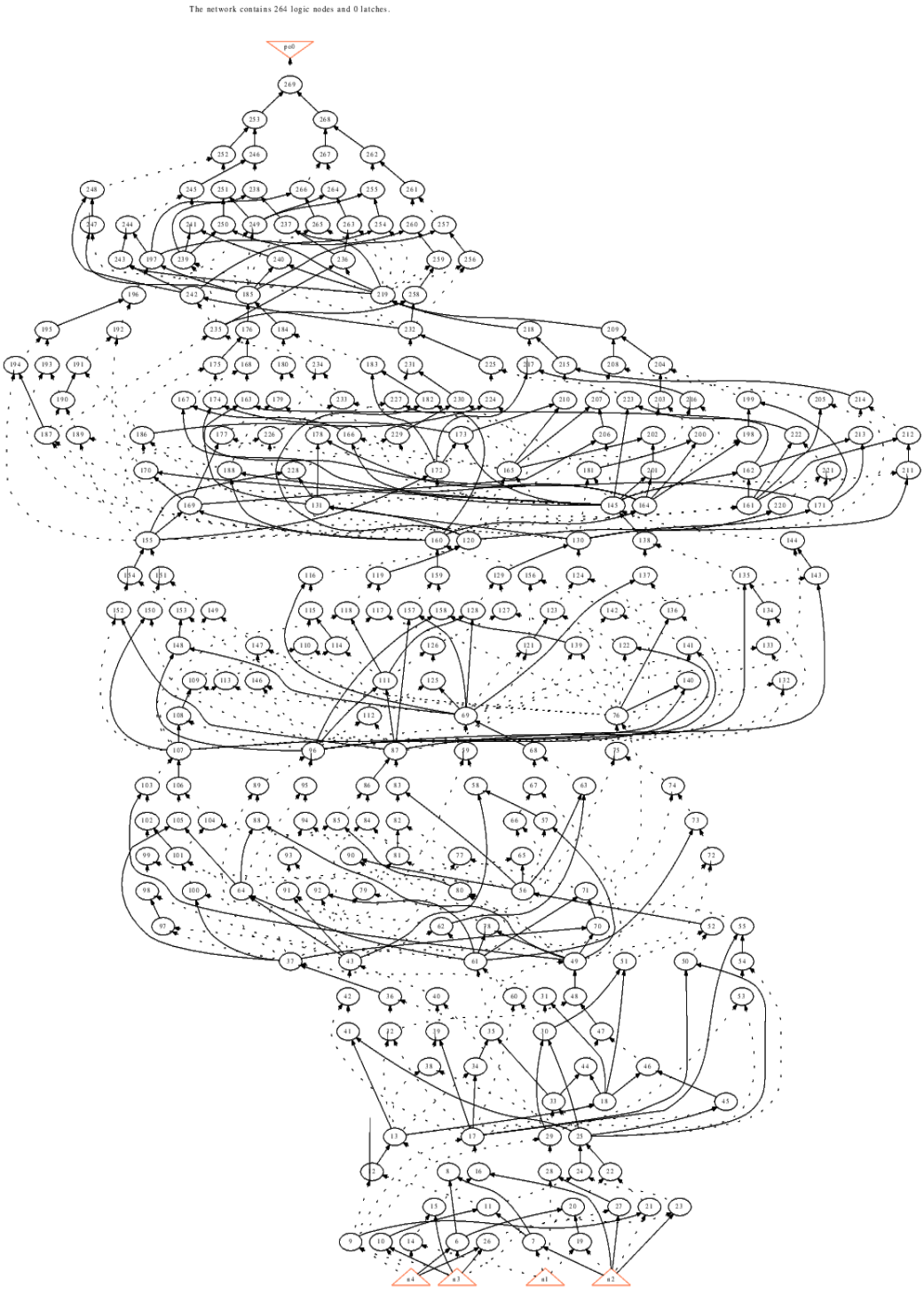


Figure 9.8: The AIG from a larger neural network (4X5X5X5X5X1 excluding biases) that was trained on the same data set eliciting fig. 9.4. Like all other AIGs derived from neural networks trained on this data, homogeneous petal width/length pairs were perfectly construed.

9.3 Ensemble of Neural Networks

Since different neural networks tend to produce mostly accurate, but different AIGs, the next step is to capture the generalization from an ensemble. While section 9.2.3 showed a 4X2X1 neural network could articulate the feature combinations for prediction, this section uses two larger neural networks (4X3X3X1 and 4X8X8X8X8X8X1) to produce a realistic amount of noise (extra nodes to articulate). Furthermore, the high accuracy found by the previous section is reinforced due to the numerous sampling done (see table 9.2).

Table 9.3 and table 9.4 show the average predictions for both neural networks and their respective AIGs. The values in the top row are the inputs tested, and their columns are the average predictions from the neural network trained on X epochs and the average predictions from the respective AIG. These tables indicate that an ensemble of neural networks (given enough epochs) can be accurately represented by an AIG. The neural network was trained on NN1, NN2, NN3, NN5, NN6, and NN7 instances (see fig. 9.4).

Figure 9.9 shows the AIG accuracy compared to the neural network confidence (how close to 0% or 100% the neural network predicts the class). This figure confirms that inaccuracies are more likely to occur when the neural network is uncertain (close to 50%). Since more training (epochs) causes neural networks to become more certain, the more trained networks yield more accurate AIGs. This training need not be enough to overfit the data. Table 9.4 and table 9.3 shows the neural network correctly generalizes for 4000 epochs and table 9.2 and fig. 9.9 show 4000 epochs make the neural network certain enough to yield highly accurate AIGs.

Table 9.2: Summary for the two neural network ensembles of dimensions 4X3X3X1 (smaller) and 4X8X8X8X8X8X1 (larger) trained on the data set mentioned in fig. 9.4.

Samples:	1000	1000	1000	750	500
Epochs:	500	1000	2000	4000	8000
Average AIG Accuracy (smaller):	0.9443	0.9604	0.9689	0.9740	0.9749
Average AIG Accuracy (larger):	0.9486	0.9532	0.9511	0.9517	0.9554

Table 9.3: Average neural network (4X3X3X1) probabilities and AIG approximations given inputs (sepal length, sepal width, petal length, petal width). Interestingly, the pairs (NN11, AIG11) and (NN14, AIG14) have a negative correlation while table 9.4 has no negative correlations.

	(0,0,0,0)	(0,1,0,0)	(1,1,0,0)	(1,0,0,0)	(1,0,1,1)	(1,1,1,1)	(0,0,1,1)	(0,1,1,1)
Epochs	NN1	NN2	NN3	NN4	NN5	NN6	NN7	NN8
500	0.17	0.12	0.15	0.46	0.87	0.84	0.85	0.71
1000	0.04	0.03	0.03	0.29	0.97	0.97	0.97	0.91
2000	0	0	0	0.08	1	1	1	0.99
4000	0	0	0	0	1	1	1	1
8000	0	0	0	0	1	1	1	1
Epochs	AIG1	AIG2	AIG3	AIG4	AIG5	AIG6	AIG7	AIG8
500	0	0	0	0.53	1	1	1	0.97
1000	0	0	0	0.23	1	1	1	1
2000	0	0	0	0.08	1	1	1	1
4000	0	0	0	0.02	1	1	1	1
8000	0	0	0	0.01	1	1	1	1

	(0,0,0,1)	(0,1,0,1)	(1,1,0,1)	(1,0,0,1)	(1,0,1,0)	(1,1,1,0)	(0,0,1,0)	(0,1,1,0)
Epochs	NN9	NN10	NN11	NN12	NN13	NN14	NN15	NN16
500	0.65	0.21	0.56	0.83	0.83	0.55	0.65	0.21
1000	0.8	0.06	0.61	0.96	0.96	0.63	0.81	0.06
2000	0.92	0.01	0.68	0.99	0.99	0.68	0.92	0.01
4000	0.97	0	0.73	1	1	0.72	0.97	0
8000	0.98	0	0.72	1	1	0.73	0.98	0
Epochs	AIG9	AIG10	AIG11	AIG12	AIG13	AIG14	AIG15	AIG16
500	0.97	0.01	0.79	1	1	0.77	0.96	0.01
1000	0.96	0	0.74	1	1	0.77	0.96	0
2000	0.97	0	0.73	1	1	0.76	0.97	0
4000	0.96	0	0.73	1	1	0.71	0.95	0
8000	0.95	0	0.7	1	1	0.71	0.95	0

Table 9.4: Average neural network (4X8X8X8X8X1) probabilities and AIG approximations given inputs (sepal length, sepal width, petal length, petal width).

	(0,0,0,0)	(0,1,0,0)	(1,1,0,0)	(1,0,0,0)	(1,0,1,1)	(1,1,1,1)	(0,0,1,1)	(0,1,1,1)
Epochs	NN1	NN2	NN3	NN4	NN5	NN6	NN7	NN8
500	0.03	0.02	0.03	0.27	0.98	0.98	0.98	0.97
1000	0.01	0.01	0.01	0.22	1	1	1	0.99
2000	0	0	0	0.19	1	1	1	1
4000	0	0	0	0.16	1	1	1	1
8000	0	0	0	0.19	1	1	1	1
Epochs	AIG1	AIG2	AIG3	AIG4	AIG5	AIG6	AIG7	AIG8
500	0	0	0	0.68	1	1	1	1
1000	0	0	0	0.63	1	1	1	1
2000	0	0	0	0.62	1	1	1	1
4000	0	0	0	0.59	1	1	1	1
8000	0	0	0	0.56	1	1	1	1

	(0,0,0,1)	(0,1,0,1)	(1,1,0,1)	(1,0,0,1)	(1,0,1,0)	(1,1,1,0)	(0,0,1,0)	(0,1,1,0)
Epochs	NN9	NN10	NN11	NN12	NN13	NN14	NN15	NN16
500	0.95	0.03	0.8	0.98	0.98	0.79	0.95	0.03
1000	0.98	0.01	0.84	1	1	0.84	0.99	0.01
2000	0.99	0	0.82	1	1	0.82	0.99	0
4000	0.99	0	0.81	1	1	0.84	0.99	0
8000	1	0	0.82	1	1	0.81	0.99	0
Epochs	AIG9	AIG10	AIG11	AIG12	AIG13	AIG14	AIG15	AIG16
500	1	0.01	0.92	1	1	0.92	1	0.01
1000	1	0.01	0.93	1	1	0.94	1	0.02
2000	1	0.01	0.91	1	1	0.91	1	0.01
4000	0.99	0.01	0.91	1	1	0.9	1	0.01
8000	1	0.01	0.92	1	1	0.91	1	0

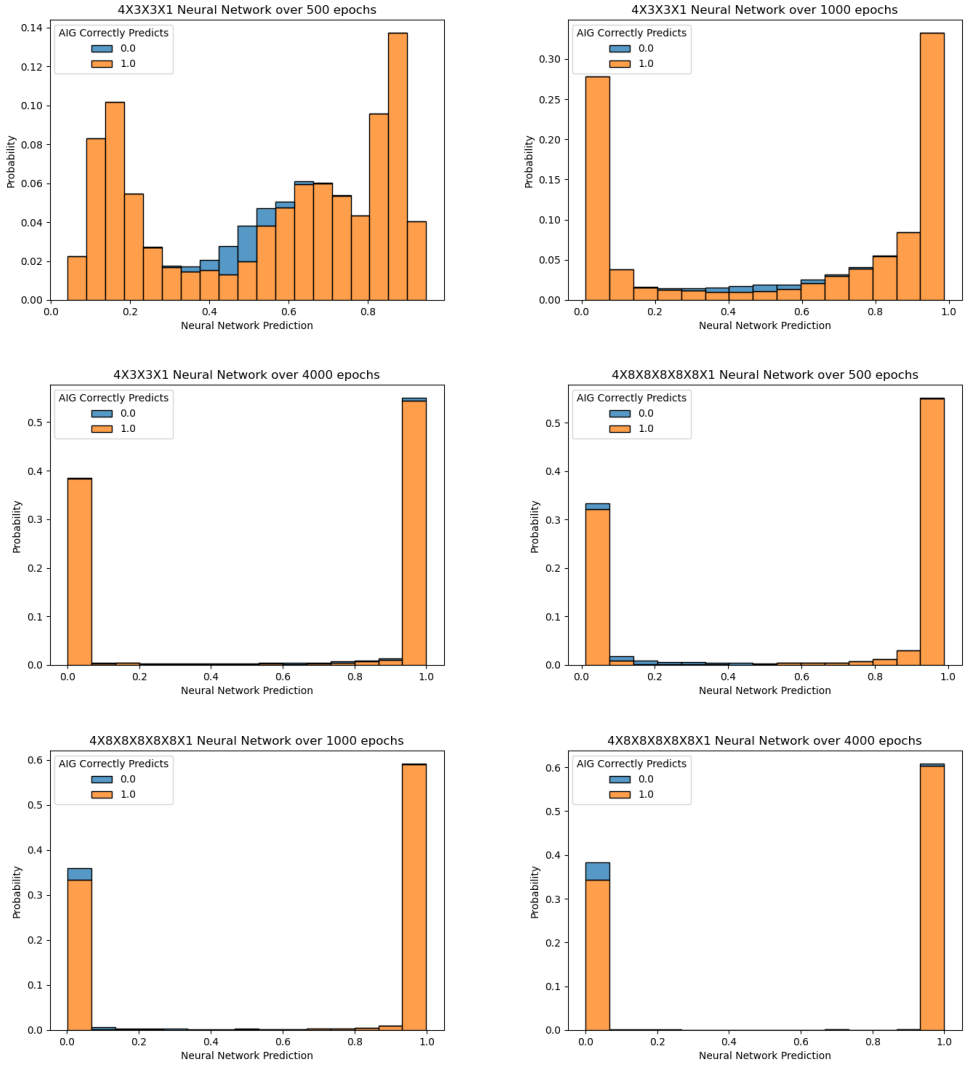


Figure 9.9: Distribution plots that shows the AIG's accuracy for all 16 inputs over all samples of the each neural network for the respective number of epochs. The blue (or orange) colors are cases were the AIG was incorrect (correct).

Part IV

Results

Chapter 10

Empirical Complexity

This chapter compares $\mathcal{B}$'s average complexity, worst-case complexity, and the alternative brute force's complexity in time and space. The average complexity was derived by creating a random array of weights with a uniform distribution. Future work can investigate the nature of neural network nodes and how close a uniformly-random distribution properly represents the average neural node. All regression fits were done with the SciPy Python library.

10.1 Node Complexity

This section is the best empirical evidence for $\mathcal{B}$'s average case of $O(2^{0.5n})$ and $\mathcal{B}$'s worst-case being $\theta(2^n)$. $\mathcal{B}$ only needs to remember the current path being traversed (memory of $O(n)$), but the sum-of-products remembered for $\mathcal{N}$ scales with $\mathcal{B}$'s. Therefore, the memory complexity of $\mathcal{B}$ is its leaf nodes whose complexities are found to be in $O(2^n)$ with an average of $O(2^{0.5})$.

10.1.1 Average Node Complexity

After analyzing fig. 10.1, we see the best-fit regression for $a \times 2^n$ is the worst fit of the three regression functions. The other two regressions (which indicate $O(2^{cn})$ for $c \approx 0.5$) are the better fit for the data.

Percent increases In fig. 10.2, the percentage increase from the average amount in the weight size prior is shown in a scatterplot. Since 10,000 out of the possible $c \times 2^n$ samples were used for each weight size, there is an

Table 10.1: Average complexities

Node Type	2^{bn}	$a \times 2^{bn}$	$a \times 2^n$
Leaves	$\approx 2^{4.743 \times 10^{-1} n}$	$\approx 1.870 \times 10^{-1} \times 2^{5.269 \times 10^{-1} n}$	$\approx 4.362 \times 10^{-8} \times 2^n$
Parents	$\approx 2^{4.956 \times 10^{-1} n}$	$\approx 6.184 \times 10^{-1} \times 2^{5.107 \times 10^{-1} n}$	$\approx 8.548 \times 10^{-8} \times 2^n$
Total Nodes	$\approx 2^{5.085 \times 10^{-1} n}$	$\approx 7.859 \times 10^{-1} \times 2^{5.160 \times 10^{-1} n}$	$\approx 1.291 \times 10^{-7} \times 2^n$

increasing variance as the weight array gets larger—increasing the samples in proportion to the exponentially increasing possibilities would likely reduce the change in variance. However, this would cause sampling to be exponentially more expensive on an already exponential algorithm.

In fig. 10.2, the median and means all show an approximate increase of 40%, which indicates the average case is approximately $O(1.4^n)$. This validates the average case of $O(2^{0.5n})$ since

$$2^{0.5n} = (2^{0.5})^n \approx 1.4^n$$

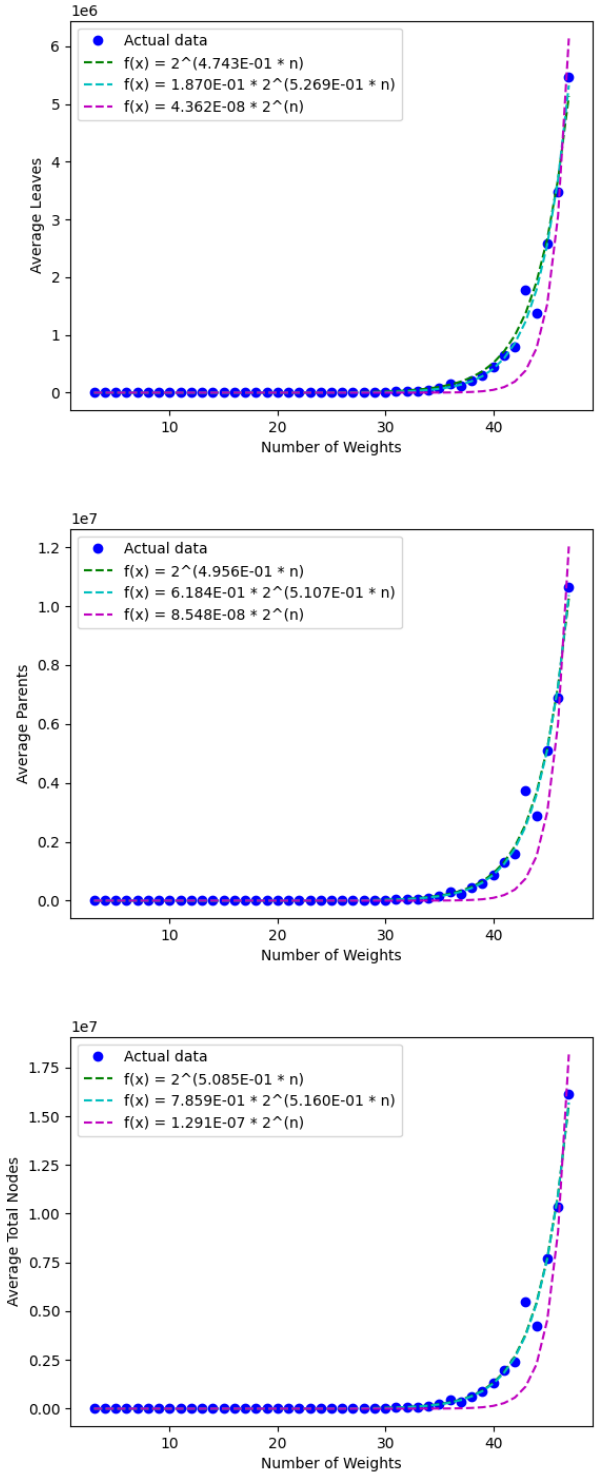
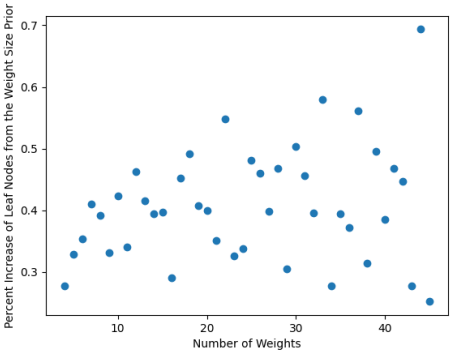
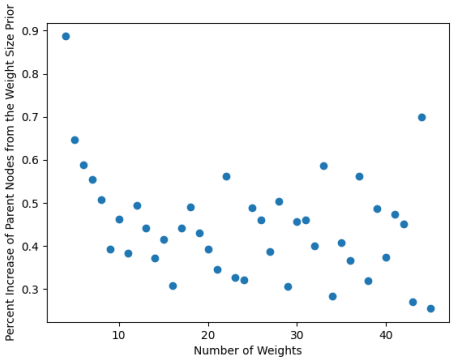


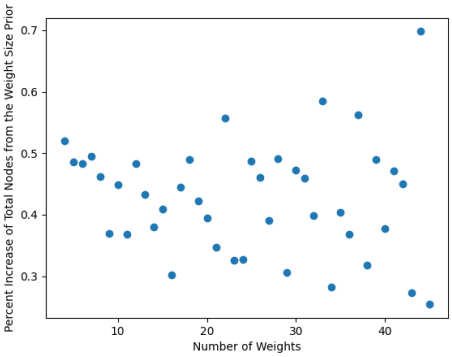
Figure 10.1: A visual representation of the average nodes and regression functions seen in table 10.1.



(a) Median = 0.398
Arithmetic mean = 0.407
Geometric mean = 0.398



(b) Median = 0.442
Arithmetic mean = 0.447
Geometric mean = 0.432



(c) Median = 0.439
Arithmetic mean = 0.427
Geometric mean = 0.418

Figure 10.2: The percentage increase for the average case (using 10,000 samples).

10.1.2 Worst-case Node Complexity

The empirical evidence in fig. 10.3 strongly supports the theoretical analysis for $\mathcal{B}$'s worst case of $O(2^n)$. In fig. 10.4, we can see alternating increases (odd versus even array sizes). Despite the odd sizes being a smaller increase from the prior even size than vice versa, both appear to be asymptotic between 0.95 and 1 which indicates a complexity of $O(2^n)$.

Table 10.2: Worst-case complexities

Node Type	2^{bn}	$a \times 2^{bn}$	$a \times 2^n$
Leaves	$\approx 2^{8.615 \times 10^{-1} n}$	$\approx 1.098 \times 10^{-1} \times 2^{9.813 \times 10^{-1} n}$	$\approx 7.770 \times 10^{-2} \times 2^n$
Parents	$\approx 2^{8.976 \times 10^{-1} n}$	$\approx 2.927 \times 10^{-1} \times 2^{9.642 \times 10^{-1} n}$	$\approx 1.509 \times 10^{-1} \times 2^n$
Total Nodes	$\approx 2^{9.201 \times 10^{-1} n}$	$\approx 3.988 \times 10^{-1} \times 2^{9.699 \times 10^{-1} n}$	$\approx 2.286 \times 10^{-1} \times 2^n$

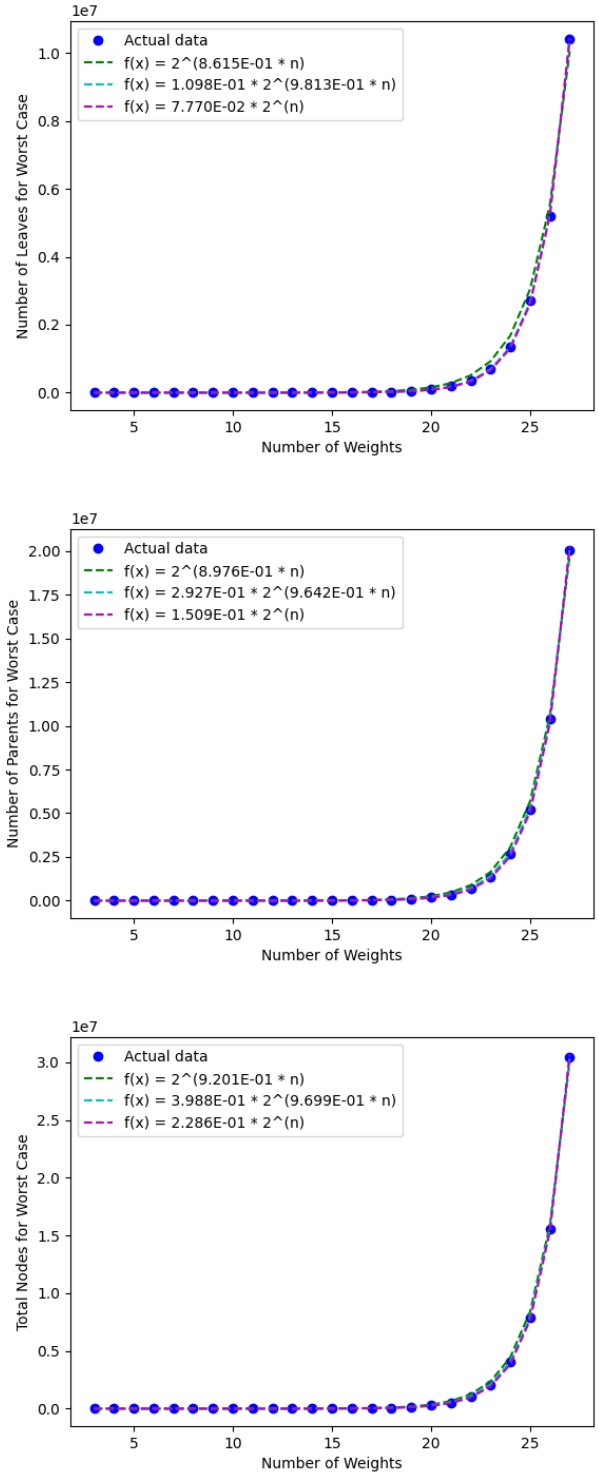


Figure 10.3: The worst-case amount of nodes with the regression functions as seen in table 10.2.

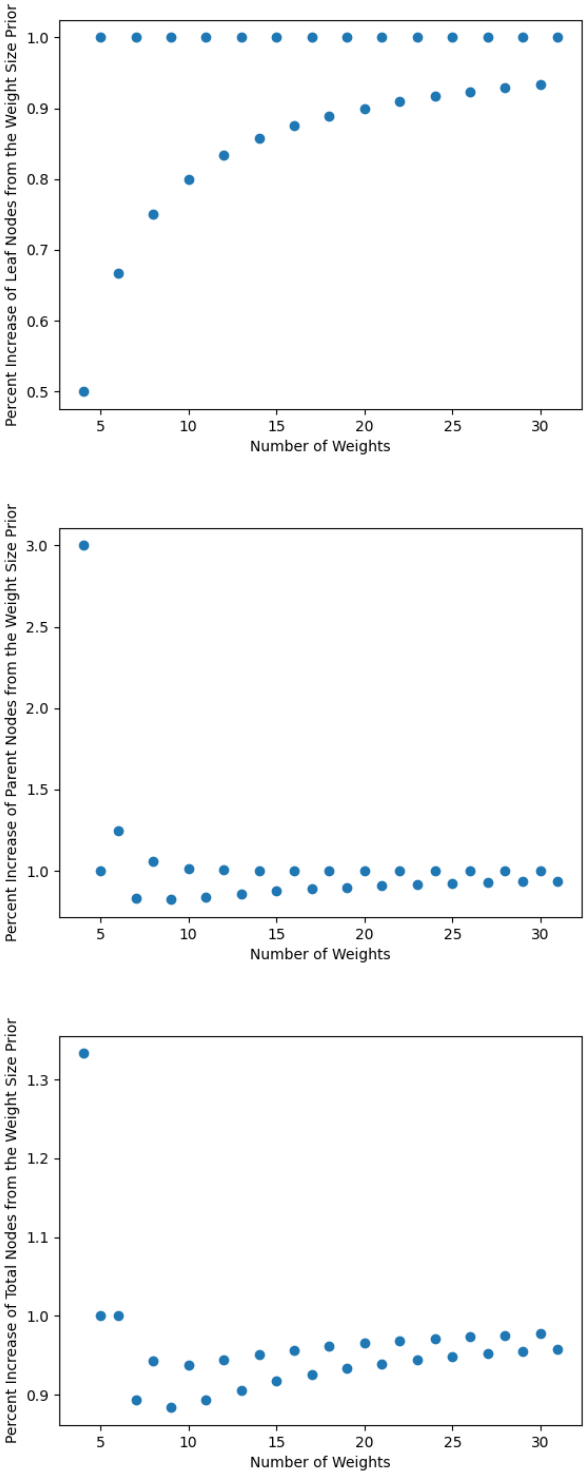


Figure 10.4: The percentage increase for the worst case.

10.1.3 Exhaustive case

By enabling early termination, the total nodes of the worst case is about $\frac{1}{9}$ of the exhaustive traversal of the tree. The exhaustive traversal ($O(2^n)$) is also faster than the brute-force approach ($O(2^n n)$) (see section 10.2.3). Table 10.3 computed percents using the coefficient a from the $a \times 2^n$ complexities in table 10.2.

Table 10.3: The exact node complexity if $\mathcal{B}$ always calculated every possibility (exhaustive case).

Node Type	Exhaustive Case	Percent of Worst-case's Complexities
Leaf	2^n	1287%
Parent	$2^n - 1$	663%
Total Nodes	$2^{n+1} - 1$	875%

10.2 Time Complexity

These results further validate complexities hypothesized in section 10.1 and give the brute force's complexity as a point of reference.

10.2.1 Average Time Complexity

Table 10.4 and fig. 10.5 use a uniformly random distribution to approximate the average case of $\mathcal{B}$. The problem of the average-case complexity here coincides with the average amount of minterms needed to express the linear combinations.

Table 10.4

Percentile	2^{bn}	$a \times 2^{bn}$	$a \times 2^n$
10th	$\approx 2^{3.270 \times 10^{-1} n}$	$\approx 6.754 \times 10^{-4} \times 2^{5.577 \times 10^{-1} n}$	$\approx 4.252 \times 10^{-10} \times 2^n$
25th	$\approx 2^{3.502 \times 10^{-1} n}$	$\approx 1.429 \times 10^{-2} \times 2^{4.845 \times 10^{-1} n}$	$\approx 8.438 \times 10^{-10} \times 2^n$
50th	$\approx 2^{3.766 \times 10^{-1} n}$	$\approx 1.224 \times 10^{-2} \times 2^{5.154 \times 10^{-1} n}$	$\approx 1.964 \times 10^{-9} \times 2^n$
75th	$\approx 2^{4.033 \times 10^{-1} n}$	$\approx 7.447 \times 10^{-4} \times 2^{6.295 \times 10^{-1} n}$	$\approx 4.743 \times 10^{-9} \times 2^n$
90th	$\approx 2^{4.384 \times 10^{-1} n}$	$\approx 2.043 \times 10^{-3} \times 2^{6.326 \times 10^{-1} n}$	$\approx 1.439 \times 10^{-8} \times 2^n$

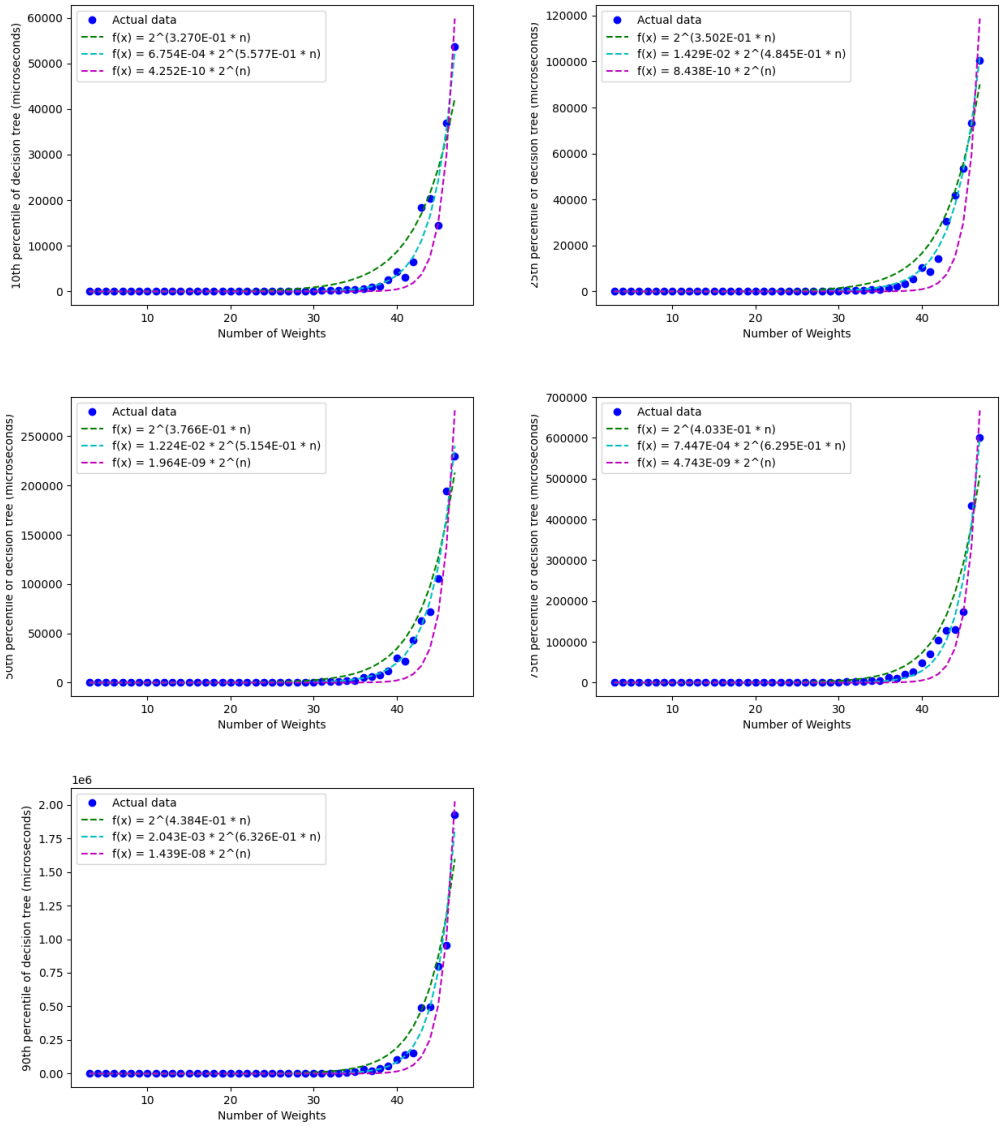


Figure 10.5

10.2.2 Worst-case Time Complexity

Table 10.5 and fig. 10.6 use weight arrays where each element has the same absolute value and the distributed between positive and negative bins are

done as evenly as possible. Figure 10.6 shows that regressions of about 2^n were the best fit. This confirms $\mathcal{B}$'s worst case of $O(2^n)$.

Table 10.5

Percentile	2^{bn}	$a \times 2^{bn}$	$a \times 2^n$
10th	$\approx 2^{7.316 \times 10^{-1} n}$	$\approx 5.530 \times 10^{-3} \times 2^{1.014n}$	$\approx 7.188 \times 10^{-3} \times 2^n$
25th	$\approx 2^{7.317 \times 10^{-1} n}$	$\approx 5.394 \times 10^{-3} \times 2^{1.016n}$	$\approx 7.206 \times 10^{-3} \times 2^n$
50th	$\approx 2^{7.319 \times 10^{-1} n}$	$\approx 5.347 \times 10^{-3} \times 2^{1.016n}$	$\approx 7.226 \times 10^{-3} \times 2^n$
75th	$\approx 2^{7.321 \times 10^{-1} n}$	$\approx 5.413 \times 10^{-3} \times 2^{1.016n}$	$\approx 7.251 \times 10^{-3} \times 2^n$
90th	$\approx 2^{7.325 \times 10^{-1} n}$	$\approx 6.205 \times 10^{-3} \times 2^{1.009n}$	$\approx 7.299 \times 10^{-3} \times 2^n$

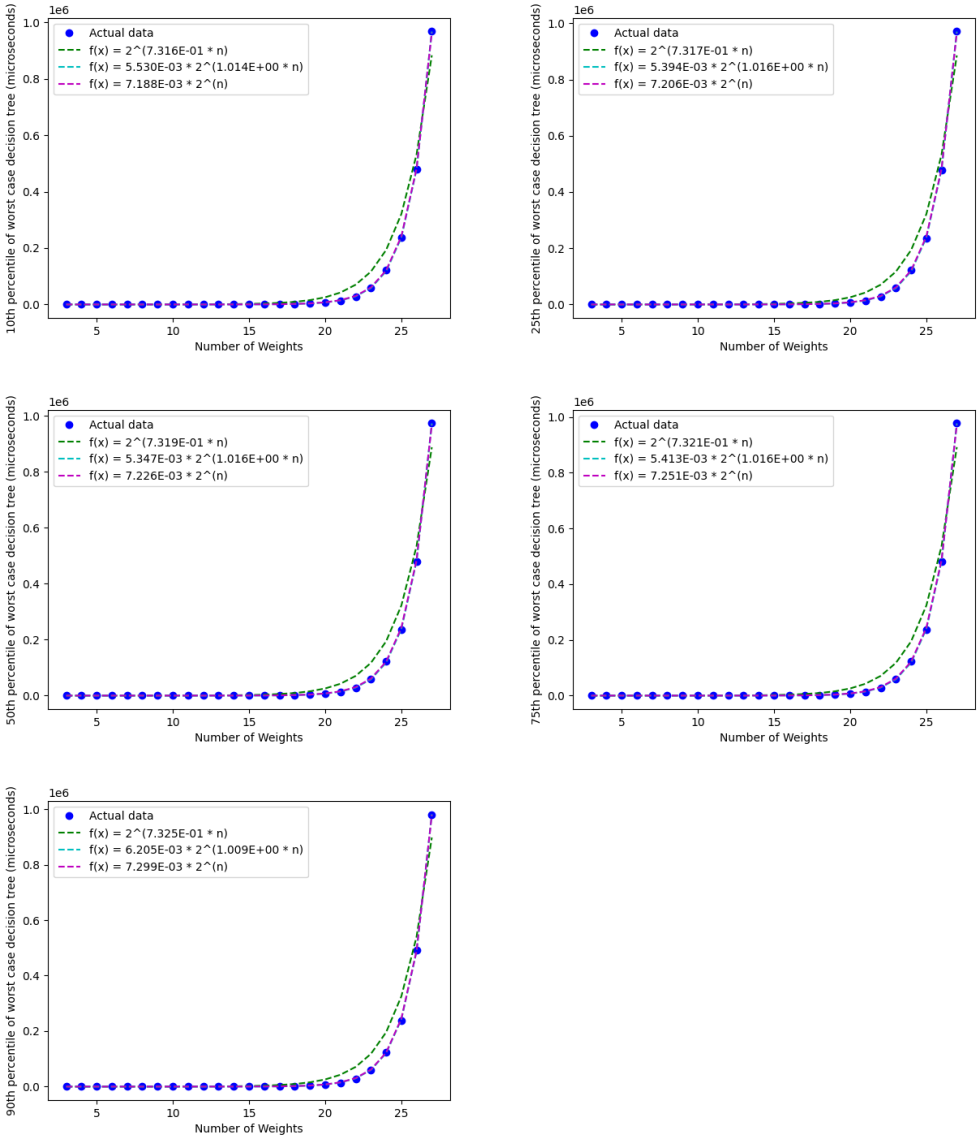


Figure 10.6

10.2.3 Brute-force complexity

Table 10.6 and fig. 10.7 approximates the complexity of a brute-force solution. The brute force solution is a simple recursive program which iterates over all

2^n mask possibilities as shown in section 10.2.3 and section 10.2.3. Following the logic from section 8.5.1, this algorithm has a complexity of $O(n \times 2^n)$. However, the regressions are done in the same manner as the prior regressions were done as a control.

Procedure initializeBruteForce

Input: n, the number of weights.
Output: productTable, a sum-of-products describing the random weight array created.

```
/* Initialize random array.                                    */
```
randArr ← array(value=random_double, size=n);
initMask ← array(value=false, size=n);
productTable ← ∅;
```
/* Let θ_t = 0 for this pseudocode.                            */
```
if *initMask* · $randArr^T > 0$ **then**
    productTable ← productTable ∪ initMask;
bruteForceIter(randArr, productTable, initMask, 0);
return *productTable*

Procedure bruteForceIter

Input:

- randArr, an array of random weight values.
- productTable, the table of products.
- currentMask, the current bitmask.
- maskIter, tracks the current position in currentMask to update.

Output: Updates the productTable which is passed by reference.

if *maskIter = currentMask.size()* **then**
  return
bruteForceIter(randArr, sopTable, currentMask, maskIter + 1);
currentMask[maskIter] = true;
if $currentMask \cdot randArr^T > 0$ **then**
  productTable ← productTable ∪ currentMask;
bruteForceIter(randArr, productTable, currentMask, maskIter + 1);
return

Table 10.6

Percentile	2^{bn}	$a \times 2^{bn}$	$a \times 2^n$
10th	$\approx 2^{8.833 \times 10^{-1} n}$	$\approx 3.381 \times 10^{-1} \times 2^{9.468 \times 10^{-1} n}$	$\approx 1.362 \times 10^{-1} \times 2^n$
25th	$\approx 2^{8.867 \times 10^{-1} n}$	$\approx 3.824 \times 10^{-1} \times 2^{9.430 \times 10^{-1} n}$	$\approx 1.444 \times 10^{-1} \times 2^n$
50th	$\approx 2^{8.947 \times 10^{-1} n}$	$\approx 5.058 \times 10^{-1} \times 2^{9.347 \times 10^{-1} n}$	$\approx 1.656 \times 10^{-1} \times 2^n$
75th	$\approx 2^{9.057 \times 10^{-1} n}$	$\approx 7.014 \times 10^{-2} \times 2^{1.061 n}$	$\approx 1.998 \times 10^{-1} \times 2^n$
90th	$\approx 2^{9.086 \times 10^{-1} n}$	$\approx 6.508 \times 10^{-2} \times 2^{1.068 n}$	$\approx 2.099 \times 10^{-1} \times 2^n$

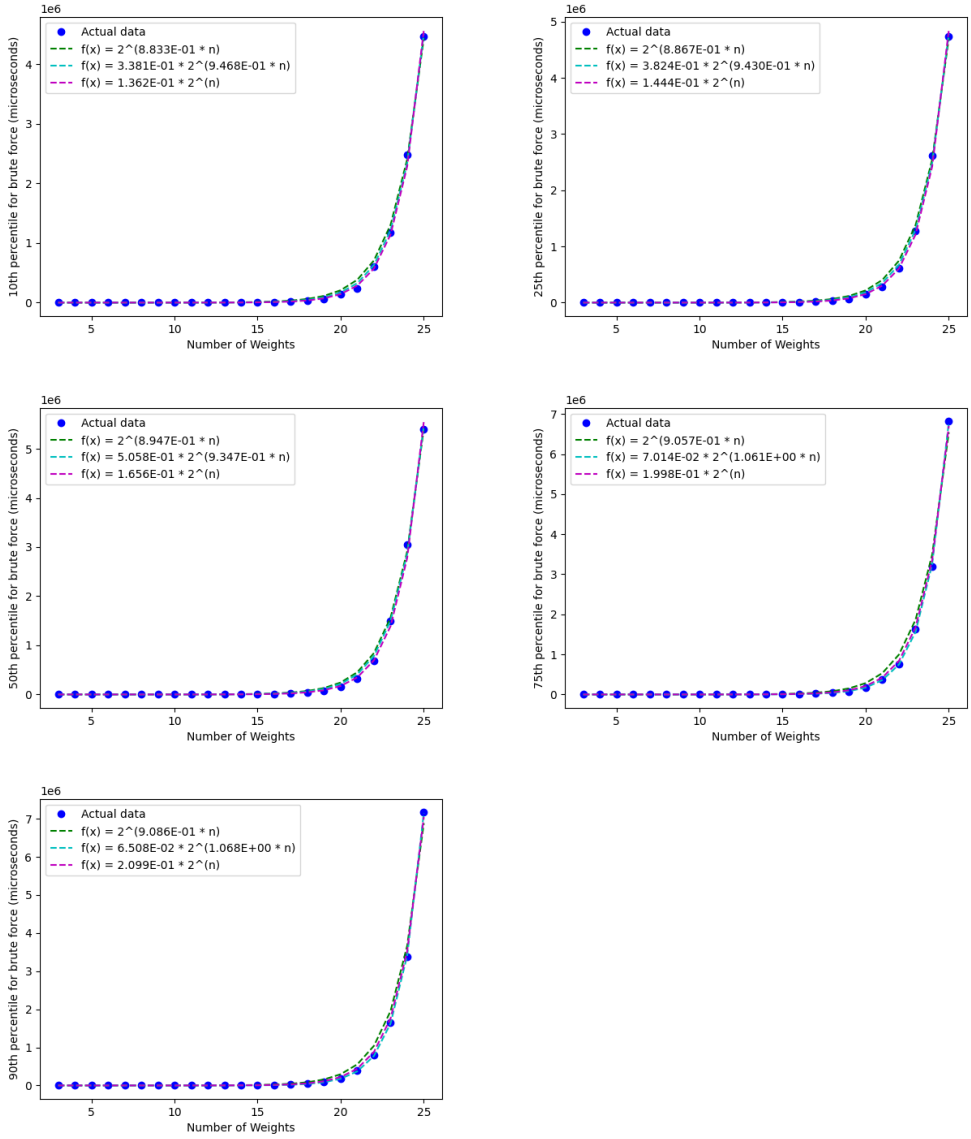

1e6
Actual data
f(x) = 2^(8.833E-01 * n)
f(x) = 3.381E-01 * 2^(9.468E-01 * n)
f(x) = 1.362E-01 * 2^(n)
10th percentile for brute force (microseconds)
Number of Weights
1e6
Actual data
f(x) = 2^(8.867E-01 * n)
f(x) = 3.824E-01 * 2^(9.430E-01 * n)
f(x) = 1.444E-01 * 2^(n)
25th percentile for brute force (microseconds)
Number of Weights
1e6
Actual data
f(x) = 2^(8.947E-01 * n)
f(x) = 5.058E-01 * 2^(9.347E-01 * n)
f(x) = 1.656E-01 * 2^(n)
50th percentile for brute force (microseconds)
Number of Weights
1e6
Actual data
f(x) = 2^(9.057E-01 * n)
f(x) = 7.014E-02 * 2^(1.061E+00 * n)
f(x) = 1.998E-01 * 2^(n)
75th percentile for brute force (microseconds)
Number of Weights
1e6
Actual data
f(x) = 2^(9.086E-01 * n)
f(x) = 6.508E-02 * 2^(1.068E+00 * n)
f(x) = 2.099E-01 * 2^(n)
90th percentile for brute force (microseconds)
Number of Weights

Figure 10.7

10.2.4 Average, Worst, and Brute Time Ratios

While the average-case and the worst-case of $\mathcal{B}$ and the brute force algorithm all have different complexities ($O(2^{0.5n}), O(2^n), O(2^n n)$, respectively), the ratios are shown below to give an idea of how much quicker $\mathcal{B}$ is at weight sizes 3 through 26. These ratios will only increase in favor of $\mathcal{B}$ for greater weight sizes.

Table 10.7: The speedups for each complexity in the form $a \times 2^n$. Note: $x : y$ is the ratio of how much faster y is than x.

Percentile	Worst-case:Average	Brute-force:Average	Brute-force:Worst-case
10th	1.69×10^8	3.21×10^8	1.89
25th	8.54×10^7	1.71×10^8	2.00
50th	3.68×10^7	8.43×10^7	2.29
75th	1.53×10^7	4.21×10^7	2.76
90th	5.07×10^6	1.46×10^7	2.88
Geometric Mean	3.34×10^7	7.78×10^7	2.33
Arithmetic Mean	6.24×10^7	1.27×10^8	2.36

Chapter 11

Results and Moving Forward

Results

The algorithms ($\mathcal{N}$ in section 8.3 and $\mathcal{B}$ in section 8.2.2) presented in this thesis made a bold abstraction of the activation nodes (section 7.2) and input values (section 9.2.3). The AIGs constructed by these algorithms were completely accurate on certain neural networks (chapter 6, section 9.1). In chapter 9, it was discovered that the AIGs were inaccurate about half of the time when most (or all) of the neural nodes had predictions at about 50% (uncertain predictions). Section 9.3 showed that the AIGs produced by $\mathcal{N}$ were highly accurate on average.

Future Work

Pruning Unnecessary intricacies arise when neural networks are larger than needed. Pruning the neural network before, during, and/or after training will mitigate bloated graphs such as fig. 9.8. See this brief survey for more information on neural network pruning: [53].

Input Space In chapter 9, we found that capturing relationships for impossible inputs could heavily complicate the end products. To remedy this, we can remove the impossible or implausible instances from the AIG's consideration. This could be done inside of $\mathcal{N}$ or by amending the layered sum-of-products produced by $\mathcal{N}$. The advantages of this idea could include a quicker algorithm and a more concise and accurate depiction of the neural network.

Related work Brudermueller et al. reduced the input space and noise by using random forests or LUTs [3].

Related work published after my experiments As mentioned in section 5.2, Shi et al. [1] published an article that cited an algorithm [45] to approximate neural nodes with a time complexity better than $\mathcal{B}$ ($O(2^{0.5n}n)$ versus $O(2^n)$). Additionally, Shi et al. mentions the fragility of neural networks and relates how robustness can be analyzed with the BDD produced by their algorithm, but they did not mention the memorization versus generalization problem mentioned by [3, 41] and seen in section 9.2. In future work, I plan to investigate intersecting several AIGs from several trained neural networks to elicit generalization.

AIG Intersection We saw in section 9.2.3 that the weights of neural networks could differ despite that the training algorithm only differed in the initial weight values. The commonality (intersection) of the many AIGs over the trained neural networks appears to elicit generalization and are the paths that AIGs consistently and accurately represent from the original neural networks. This intersection will highlight the best relationships while omitting the more subtle or developing relationships (see section 9.3 for supporting data). Furthermore, this intersection will cause the AIGs to be more concise which helps with comprehensibility (chapter 4).

End Product The and-inverter graph was chosen due to its eminence in theoretical Boolean operations. Other graphs may be more readable or result in fewer nodes and transitions (e.g. majority-inverter graph ([20]) or binary decision diagrams (section 3.1)).

Satisfiability This thesis also introduces a new algorithm to predict satisfiability given a data set. Step 1: train several neural networks; step 2: convert the several networks to a Boolean structure (perhaps with AIG intersections); finally, check for satisfiability using formal methods on the constructed structure.

Multiple Outputs One could binarize output functions such as softmax to approximate neural networks with multiple outputs. However, the AIG or other Boolean structure may give more than one answer per input.

Appendix A

Definitions

Abbreviations

- $\mathcal{B}$: the optimized, binary decision tree that approximates a neural node (section 8.2.2).
- $\mathcal{N}$: the algorithm that parses the neural network with $\mathcal{B}$ in reverse (algorithm 7).
- NN: neural network.
- LUT: lookup table.
- SoP: sum-of-products.
- AIG: and-inverter graph (section 3.1).
- MIG: majority-inverter graph (section 3.1).
- BDD: binary decision diagram (section 3.1).
- $\mathbb{R}$: the set of real numbers.
- $\mathbb{B} = \{0, 1\}$: the Boolean set.

Mathematical symbols

- $\leftarrow$: assignment.
- $\in$: is an element of (also read as "in").

- $\cup$: union.
- $\cap$: intersection.
- $\wedge$: and.
- $\vee$: or.
- $\neg$: not.
- $\mapsto$: maps to.
- $\rightarrow$: implies.
- $\longleftrightarrow$: bi-implication (read as "if and only if").
- $\forall$: the universal quantifier (read as "for all").
- $\exists$: the existential quantifier (read as "there exists").
- $\models$: models (or satisfies).

Complexity symbols

- O: Big-O notation—the asymptotic upper bound.
- Ω: Omega notation—the asymptotic lower bound.
- Θ: Theta notation—the asymptotic upper and lower bound.

Other

- N^{th}-order relationship: a loose definition conveying how nested the relationship is (see chapter 4).
- Satisfiability: if some structure s holds for some property ϕ, then this structure satisfies (models) the property. Formally, $s \models \phi$.

Bibliography

[1] W. Shi, A. Shih, A. Darwiche, and A. Choi, "On tractable representations of binary neural networks," 04 2020.

[2] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," vol. 6174, pp. 24–40, 07 2010.

[3] T. Brudermueller, D. Shung, L. Laine, A. Stanley, S. Laursen, H. Dalton, J. Ngu, M. Schultz, J. Stegmaier, and S. Krishnaswamy, "Making logic learnable with neural networks," 02 2020.

[4] R. Guidotti, A. Monreale, F. Turini, D. Pedreschi, and F. Giannotti, "A survey of methods for explaining black box models," *ACM Computing Surveys*, vol. 51, 02 2018.

[5] R. Andrews, J. Diederich, and A. Tickle, "Survey and critique of techniques for extracting rules from trained artificial neural networks," *Knowledge-Based Systems*, vol. 6, pp. 373–389, 12 1995.

[6] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, and G. De Micheli, "The EPFL logic synthesis libraries," Nov. 2019. arXiv:1805.05121v2.

[7] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*, pp. 85–103, Springer, 1972.

[8] T. H. Abraham, "(physio) logical circuits: The intellectual origins of the mcculloch–pitts neural networks," *Journal of the History of the Behavioral Sciences*, vol. 38, no. 1, pp. 3–25, 2002.

[9] Minsky and Papert, *Perceptrons*. 1969.

[10] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[11] G. P. Zhang, "Neural networks for classification: a survey," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 30, no. 4, pp. 451–462, 2000.

[12] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.

[13] E. Fiesler, "Neural network formalization," tech. rep., IDIAP, 1992.

[14] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. The MIT Press, 2016.

[15] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *CoRR*, vol. abs/1710.05941, 2017.

[16] Y. LeCun *et al.*, "Generalization and network design strategies," *Connectionism in perspective*, vol. 19, pp. 143–155, 1989.

[17] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural networks: Tricks of the trade*, pp. 9–48, Springer, 2012.

[18] J. Hawkins, *On Intelligence: How a New Understanding of the Brain Will Lead to the Creation of Truly Intelligent Machines*. 2005.

[19] W. Duch and N. Jankowski, "Survey of neural transfer functions," *Neural Computing Surveys*, vol. 2, no. 1, pp. 163–212, 1999.

[20] L. Amarú, P. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2014.

[21] J. Qin, "Logic simulator for bench format."

[22] R. Descartes, *Rules for the Direction of the Mind*, vol. 1, p. 7–78. Cambridge University Press, 1985.

[23] J. Shlens, "A tutorial on principal component analysis," *arXiv preprint arXiv:1404.1100*, 2014.

[24] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[25] E. B. Goldstein and J. Brockmole, *Sensation and Perception*. Cengage Learning, 2016.

[26] Y. Danilov and M. Tyler, "Brainport: an alternative input to the brain," *Journal of integrative neuroscience*, vol. 4, no. 04, pp. 537–550, 2005.

[27] C. A. Seger, "Implicit learning.," *Psychological bulletin*, vol. 115, no. 2, p. 163, 1994.

[28] G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information.," *Psychological review*, vol. 63, no. 2, p. 81, 1956.

[29] S. Feferman, "Tarski's conceptual analysis of semantical notions," *New essays on Tarski and philosophy*, pp. 72–93, 2008.

[30] C. Smorynski, "The incompleteness theorems," in *Studies in Logic and the Foundations of Mathematics*, vol. 90, pp. 821–865, Elsevier, 1977.

[31] W. Timberlake, "Behavior systems, associationism, and pavlovian conditioning," *Psychonomic Bulletin & Review*, vol. 1, no. 4, pp. 405–420, 1994.

[32] J. A. Fodor, Z. W. Pylyshyn, *et al.*, "Connectionism and cognitive architecture: A critical analysis," *Cognition*, vol. 28, no. 1-2, pp. 3–71, 1988.

[33] J. Sutton, *Philosophy and memory traces: Descartes to connectionism*. Cambridge University Press, 1998.

[34] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London mathematical society*, vol. 2, no. 1, pp. 230–265, 1937.

[35] J. Nash, "Non-cooperative games," *Annals of mathematics*, pp. 286–295, 1951.

[36] C. F. Camerer, *Behavioral game theory: Experiments in strategic interaction*. Princeton University Press, 2011.

[37] J. W. Weibull, *Evolutionary game theory*. MIT press, 1997.

[38] N. Eén and N. Sörensson, "An extensible sat-solver," in *International conference on theory and applications of satisfiability testing*, pp. 502–518, Springer, 2003.

[39] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *Proceedings of the 38th annual Design Automation Conference*, pp. 530–535, 2001.

[40] E. Goldberg and Y. Novikov, "Berkmin: A fast and robust sat-solver," *Discrete Applied Mathematics*, vol. 155, no. 12, pp. 1549–1561, 2007.

[41] S. Chatterjee and A. Mishchenko, "Circuit-based intrinsic methods to detect overfitting," *arXiv preprint arXiv:1907.01991*, 2019.

[42] B. Bünz and M. Lamm, "Graph neural networks and boolean satisfiability," *arXiv preprint arXiv:1702.03592*, 2017.

[43] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in neural information processing systems*, pp. 4107–4115, 2016.

[44] W. J. Murdoch, C. Singh, K. Kumbier, R. Abbasi-Asl, and B. Yu, "Interpretable machine learning: definitions, methods, and applications," *arXiv preprint arXiv:1901.04592*, 2019.

[45] H. Chan and A. Darwiche, "Reasoning about bayesian network classifiers," *arXiv preprint arXiv:1212.2470*, 2012.

[46] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[47] J. Ellson, E. Gansner, L. Koutsofios, S. North, G. Woodhull, S. Description, and L. Technologies, "Graphviz — open source graph drawing tools," in *Lecture Notes in Computer Science*, pp. 483–484, Springer-Verlag, 2001.

[48] A. Biere, K. Heljanko, and S. Wieringa, "AIGER 1.9 and beyond," Tech. Rep. 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.

[49] M. Vazquez-Chanlatte, "mvcisback/py-aiger," Aug. 2018.

[50] R. Raz, "On the complexity of matrix product," in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pp. 144–151, 2002.

[51] J. Alman and V. V. Williams, "A refined laser method and faster matrix multiplication," 2020.

[52] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning," *Nature communications*, vol. 5, no. 1, pp. 1–9, 2014.

[53] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag, "What is the state of neural network pruning?," 2020.