Reducing the Neural Network Traversal Space for Boolean Abstractions

Jarren Briscoe[1] , **Brian Rague**[1] , **Third Author**[2,3] , **Robert Ball**[1]

[1]Weber State University
[2]Second Affiliation
[3]Third Affiliation
jarrenbriscoe@gmail.com, {brague, robertball}@weber.edu

Abstract

The inherent intricate topology of a neural network (NN) decreases our understanding of its function and purpose. Neural network abstraction and analysis techniques are designed to increase the comprehensibility of these computing structures. To achieve a more concise and interpretable representation of a NN as a Boolean graph (BG), the Neural Constantness Heuristic, Neural Constant Propagation, shared logic, and negligible neural nodes are considered to reduce a neural layer's input space and reduce the number of approximated neural nodes. Two parsing methods to translate NNs to BGs, reverse traversal ($\mathcal{N}$) and forward traversal ($\mathcal{F}$), are contrasted. For most use cases, $\mathcal{N}$ is the better choice.

1 Introduction

1.1 Background and Purpose

Neural networks (NNs) are a powerful machine learning tool, yet semantically they are labyrinthine. (Briscoe 2021) found the most essential attribute lacking in the design and formulation of NNs is conciseness. This explains why many authors have opted to use Boolean graphs (BGs) as an explanatory medium (Briscoe 2021; Brudermueller et al. 2020; Shi et al. 2020; Choi et al. 2017; Chan and Darwiche 2012).

This intrinsic incoherent nature of NNs has inhibited the widespread adoption of these computational structures primarily due to the lack of accountability (Kroll et al. 2016), liability (Kingston 2016), and safety (Danks and London 2017) when applied to sensitive tasks such as predicting qualified job candidates and medical treatments. Furthermore, some legal guidelines distinctly prohibit black-box decisions to prevent potential discrimination (such as Recital 71 EU DGPR of the EU general data protection regulation). Additionally, safety-critical applications have avoided neural networks because of their inaccessibility to feasible formal verification techniques. For example, neural-network-driven medical devices or intrusion prevention systems on networks required to maintain accessibility (such as military communication) lack formal verification and have serious risks. This work allows formal verification for *approximations* of neural networks.

Neural Networks Neural networks train on a data set containing inputs and classifications (supervised learning), unlabelled data (unsupervised learning), or by policies (reinforcement learning). After training, the neural network predicts classifications for new inputs. Since neural network structures and algorithms are quite diverse, in this paper we limit the scope of neural networks to the most common type. Specifically, they are fully interlayer-connected, symmetric, first-order neural networks with a finite amount of layers and nodes and binary input (Figure 2). Despite training on a binary data set, the activation functions are real-valued. This work can be extended to other neural networks such as binary convolutional neural networks (Choi et al. 2017; Shi et al. 2020). For detailed NN terminology and formal definitions, see *Neural Network Formalization* (Fiesler 1992).

Explaining Neural Networks From many techniques attempting to conceptualize and explain neural networks (Guidotti et al. 2018; Andrews, Diederich, and Tickle 1995; Baehrens et al. 2010; Brudermueller et al. 2020; Shi et al. 2020; Choi et al. 2017; Briscoe 2021), three generic categories emerge: decompositional, eclectic, and pedagogical.

Decompositional techniques consider individual weights, activation functions, and the finer details of the network (local learning). For example, characterizing each node of the NN by its weights is a decompositional technique.

On the other hand, a pedagogical approach discovers through an oracle or teacher (global learning). This approach allows underlying machine learning structure(s) to be versatile—this technique will remain viable if the neural network black-box is replaced with a support-vector machine (SVM). A simple example of a pedagogical representation is a decision tree whose training inputs are fed into a NN, and the decision tree's learned class values are the output of the NN.

Finally, eclectic approaches combine the decompositional and pedagogical techniques. An example of an eclectic approach is directing a learning algorithm over neural nodes (decompositional and local) while determining and omitting inputs negligible to the final result (pedagogical and global).

This work presents and contrasts two neural-network parsing methods to translate neural networks to more interpretable and formally verifiable BGs. The forward traversal, $\mathcal{F}$ is strictly decompositional while $\mathcal{N}$ is mostly decompositional with some pedagogical aspects (eclectic).

1.2 Approximating the Neural Node

The simplest and least efficient method to approximate neural nodes is the brute-force method with $\Theta(2^n n)$ complexity (Algorithm 2) where n is the umber of in-degree weights for a given node. Fortunately, there are several far more efficient methods. The exponentially upper-bound methods described below yield higher accuracy than the pseudo-polynomial methods that follow.

Exponential Methods An exponential method targeting Bayesian network classifiers reports a complexity of $O(2^{0.5n} n)$ (Chan and Darwiche 2012). While this approximation was done for Bayesian network classifiers, the correlation to neural node approximation can be formalized (Choi et al. 2017). Another algorithm to approximate neural nodes exhibits a complexity of $O(2^n)$ (Briscoe 2021). From this current work, $\mathcal{N}$ improves the upper-bound exponential complexity of Chan and Darwhiche's algorithm as applied to neural networks (Choi et al. 2017) with no loss in accuracy. Assuming a certain percentage (x) of nodes are eligible for the Neural Constantness Heuristic, the new complexity becomes $O(2^{(0.5(1-x)n)} n)$.

Pseudo-Polynomial Methods Faster methods that compute a neural node with less accuracy produce a pseudo-polynomial $O(n^2 W)$[1] complexity where

$$W = |\theta_{\mathcal{D}}| + \sum_{w \in node_{i,j}} |w|$$

where $\theta_{\mathcal{D}}$ is the threshold in the activation function's domain and node$_{i,j}$ is a vector of in-degree weights. Furthermore, these weights must be integers with fixed precision (Shi et al. 2020).

If one uses a constant maximum depth d for the binary decision tree (Briscoe 2021), then the entire *neural network* can be computed in $O(n)$ time where n is the number of neural nodes. This is a pseudo-polynomial complexity since the constant d causes the individual neural node approximation to be a constant of $O(2^d) = O(1)$ and is still $O(2^n)$ for $n < d$. Of course, if d is not constant, then the neural node complexity becomes $O(2^{\min(n,d)})$. I am currently working on creating a flexible maximum depth that considers the weight vector's distribution and size. Future work could extend this idea to Chan and Darwhiche's algorithm on Bayesian Network classifiers and reduce the $O(2^{0.5n} n)$ complexity with minimal accuracy loss (and as employed by (Choi et al. 2017)). Enforcing a maximum depth is equivalent to capping a maximum amount of weights considered, and since the weights' importance are ordered by the greatest absolute value (Briscoe 2021), the weight with the largest absolute values should be considered first.

$\mathcal{B}$ Generalization For the sake of brevity, the description of the steps to approximate neural nodes will be omitted and will henceforth be generalized as $\mathcal{B}$. For walkthroughs, I will use the simple brute-force method $\Theta(2^n n)$ (Algorithm 2) and introduce a Neural Constantness Heuristic $\Theta(n)$ (Section 3.2).

Final Product The final translation of the NN can be any Boolean structure. Shi et al. and Choi et al. used Ordered Binary Decision Diagrams (OBDDs) and Sentential Decision Diagrams (SDDs) while Brudermueller et al. and Briscoe employed And-Inverter Graphs (AIGs) (Shi et al. 2020; Choi et al. 2017; Briscoe 2021; Brudermueller et al. 2020). However, other Boolean structures exist with different topological and/or operational features that highlight specific design (e.g. comprehensibility, Boolean optimizations, and hardware optimizations). The final product for this work will be abstracted as a BG (Boolean Graph).

1.3 Format

This paper is organized as follows: Section 2 generalizes allowable activation functions and thresholds for parsing algorithms; Section 3 defines and gives examples of the Neural Constantness Heuristic, the Neural Constant Propagation, $\mathcal{F}$, and $\mathcal{N}$; Section 4 discusses the weight-space and node-space reductions used in Section 3; finally, Section 5 gives a summary and addresses future work.

2 Activation Functions

Shi and Choi only considered ReLU and sigmoid functions (Shi et al. 2020; Choi et al. 2017). However, the properties of activation functions for successful resolution to true/false must allow a reasonable Boolean cast (Conjunctive Limit Constraint), and conversion to a binary step function (Diagonal Quadrants Property).

Conjunctive Limit Constraint:

$$\lim_{\theta \to -\infty} f(\theta) = 0 \text{ and } \lim_{\theta \to \infty} f(\theta) > 0 \qquad (1)$$

The conjunctive limit constraint allows us to treat one side of the range threshold as false (analogous to zero) and the other side as true (analogous to a positive number). Functions that have negative and positive limits (such as $\tanh$) are better defined within an inhibitory/excitatory context. It can be experimentally shown that the accuracy for NN to BG methods decrease for functions like $\tanh$. The Swish activation function (Ramachandran, Zoph, and Le 2017) lies in between the inhibitory/excitatory and true/false spectrums since there are negative range values but it still satisfies the Conjunctive Limit Constraint. As such, the accuracy of approximating neural networks with the Swish function is likely between the lower accuracy of $\tanh$ and at least as accurate ReLU.

Diagonal Quadrants Property Converting a real-valued function ($f_{\mathbb{R}}$) to a binary-step function ($f_{\mathbb{B}}$) is best suited for real-valued functions that only lie in two diagonal quadrants

[1] While the original text only mentions the $O(nW)$ complexity to compile the OBDD (Shi et al. 2020), the complexity to compute the OBDD is $O(n^2 W)$.

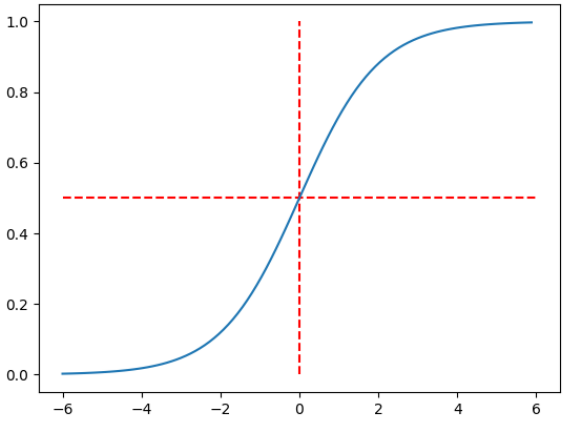


Figure 1: $\sigma(\theta)$ with a threshold of $\sigma(0) = 0.5$ satisfies the diagonal quadrants property.

or on the boundaries, centered on the threshold $f_{\mathbb{R}}(\theta_{\mathcal{D}}) = \theta_{\mathcal{R}}$. Consider the sigmoid function $\sigma_{\mathbb{R}}(\theta) = \frac{1}{1+e^{-\theta}}$. When considering $\sigma_{\mathbb{R}}$'s range $\mathcal{R} \in (0, 1)$, we want the range's threshold $\theta_{\mathcal{R}}$ to be 0.5 which corresponds to $\theta_{\mathcal{D}} = 0$ on the domain: $\sigma_{\mathbb{R}}(0) = 0.5$. Drawing one horizontal and one vertical line through $\sigma_{\mathbb{R}}(0) = 0.5$ creates the quadrants. Since $\sigma_{\mathbb{R}}$ with threshold $\sigma_{\mathbb{R}}(0) = 0.5$ is contained in diagonal quadrants (top-right and bottom-left—see Figure 1), this activation function is suitable for conversion to a binary-step function $\sigma_{\mathbb{B}}$. With the threshold $\theta_{\mathcal{R}}$:

$$\sigma_{\mathbb{B}}(\theta; \sigma_{\mathbb{R}}, \theta_{\mathcal{R}}) = \begin{cases} 1 & \text{if } \sigma_{\mathbb{R}}(\theta) \geq \theta_{\mathcal{R}}, \\ 0 & \text{otherwise.} \end{cases}$$

For monotonic functions, any threshold satisfies the diagonal quadrant property. However, a reasonable threshold choice increases the accuracy of the conversion algorithm—if one chooses the threshold $\sigma_{\mathbb{R}}(-10000) \approx 0$ then the BG will almost certainly always yield true.

Threshold on the Domain ($\theta_{\mathcal{D}}$) Since this work is limited to activation functions satisfying the Diagonal Quadrant Property, the threshold can be uniquely identified by only considering the domain. Once the domain's threshold is found, the activation function and range threshold can be effectively discarded. The more computationally efficient (and equivalent) definition of $\sigma_{\mathbb{B}}$ is then:

$$\sigma_{\mathbb{B}}(\theta; \theta, \theta_{\mathcal{D}}) = \begin{cases} 1 & \text{if } \theta \geq \theta_{\mathcal{D}}, \\ 0 & \text{otherwise.}^{2} \end{cases}$$

Henceforth, $\theta_{\mathcal{D}} = 0$ and when $\sigma_{\mathbb{B}}$ is used without all parameters, the implicit parameters will be $(\theta; \theta, 0)$.

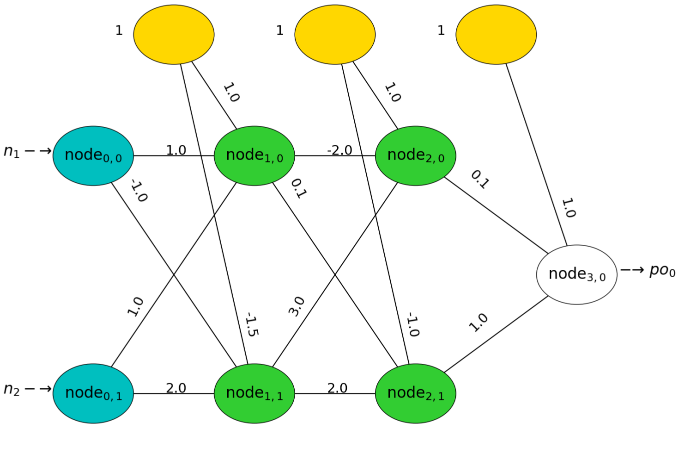
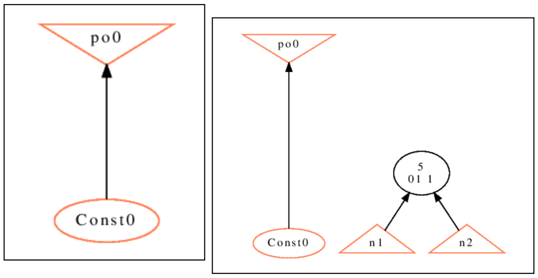


Figure 2: The neural network for the use case presented in Figure 3. $\mathcal{N}$ determines $po_0 = true$ by only parsing the output node with three weights. The hidden nodes (green) and the output node (white) use the σ activation function. Additionally, the input nodes are in the leftmost layer (cyan) and the bias nodes are in the top row (yellow).



(a) Simplified BG abstraction of Figure 2 using $\mathcal{N}$.

(b) Simplified BG abstraction of Figure 2 using $\mathcal{F}$.

Figure 3

3 Neural Network Traversal

3.1 Final Products

This section introduces the Neural Constantness Heuristic and contrasts two network parsing techniques; the forward traversal ($\mathcal{F}$) and the reverse traversal ($\mathcal{N}$).

When parsing the neural network in Figure 2, $\mathcal{N}$ only parses the output node with three weights while $\mathcal{F}$ parsed all hidden nodes and the output node for a total of fifteen weights. Additionally, it was found that $\mathcal{N}$'s BG was more comprehensible and less prone to machine or human misinterpretation than $\mathcal{F}$'s BG.

3.2 Algorithms

Neural Network Data Structure The neural network, NN, is defined as an array of layers.

$$NN \leftarrow [l_1, l_2, \ldots, l_m] \tag{2}$$

Each layer, l_i, is an array of nodes.

$$l_i \leftarrow [\text{node}_{i,0}, \text{node}_{i,1}, \ldots, \text{node}_{i,n}] \tag{3}$$

[2] Since $\sigma_{\mathbb{B}}(\theta_{\mathcal{D}})$ is undefined, letting $\sigma_{\mathbb{B}}(\theta_{\mathcal{D}}) = 0$ or $\sigma_{\mathbb{B}}(\theta_{\mathcal{D}}) = 1$ is arbitrary or dependent on special cases.

Each node, node$_{i,j}$, is an array of in-degree weights where $w_{i,j,b}$ is the bias weight and $w_{i,j,k}$ is the weight from node$_{i-1,k}$ to node$_{i,j}$.

$$\text{node}_{i,j} \leftarrow [w_{i,j,b}, w_{i,j,0}, w_{i,j,1}, \ldots, w_{i,j,p}] \tag{4}$$

In the case of where the previous nodes is $p+1$ NN is a 2D matrix of nodes and a 3D matrix of in-degree weights. In the context of the current node$_{i,j}$, the bias weight is expressed as:

$$b = w_{i,j,b}.$$

Output Definitions out$_i$ is the array of binarized outputs for layer l_i and $o_{i,j}$ is the binarized output of node$_{i,j}$.

$$\text{out}_i \leftarrow [o_{i,0}, o_{i,1}, \ldots, o_{i,n}] \tag{5}$$

Since the neural network uses the σ activation function (Figure 1), we use the $\sigma_{\mathbb{B}}$ binary-step function as defined in Section 2 and the threshold $\theta_{\mathcal{D}} = 0$. Note that the θ in activation functions $f(\theta)$ is actually a dependent variable:

$$\theta = \theta(\vec{x}, \vec{w}, b) = \vec{x} \cdot \vec{w} + b$$

where $\vec{w}^T$ is the transpose of $\vec{w}$.

The inputs $\vec{x} = \text{out}_{i-1}$, the weights $\vec{w} = \text{node}_{i,j}$, and the bias $b = w_{i,j,b}$.

$$o_{i,j} = \sigma_{\mathbb{B}}(\theta(\text{out}_{i-1}, \text{node}_{i,j} \setminus b, b)) = \begin{cases} 1 & \text{if } \theta \geq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

Finally, out$_0$ consists of binary inputs for the neural network.

Neural Constantness Heuristic The Neural Constantness Heuristic checks for a neural node's constantness with linear computational complexity ($\Theta(n)$) where n is the number of in-degree weights for a given node. To understand this heuristic, let us create some variables and functions. For node$_{i,j}$:

- Let b be the bias weight equal to $w_{i,j,b}$.
- Let W^+ be the set of weights greater than zero in node$_{i,j} \setminus b$
- Let GV$_{i,j}$ be the largest possible combination of weights in node$_{i,j}$.
$$\text{GV}_{i,j} = b + \sum_{w \in W^+} w$$
- Let W^- be the set of weights less than zero in node$_{i,j} \setminus b$.
- Let SV$_{i,j}$ be the smallest possible combination of weights in node$_{i,j}$.
$$\text{SV}_{i,j} = b + \sum_{w \in W^-} w$$

As seen above, the associated input for the bias weight (b) is always true and must be considered when computing both GV$_{i,j}$ and SV$_{i,j}$. A heuristic to find the binarized output for a neural node is then:

$$o_{i,j} \leftarrow \begin{cases} 1 & \text{if SV}_{i,j} \geq 0, \\ 0 & \text{if GV}_{i,j} \leq 0 \\ \mathcal{B} & \text{otherwise.} \end{cases}$$

The logic is if the largest possible value of θ is less than zero, then $\sigma_{\mathbb{B}}(\theta) = 0$ for all possible θ values of node$_{i,j}$. Conversely, if the smallest possible value of θ is greater than or equal to zero, then $\sigma_{\mathbb{B}}(\theta) = 1$ for all possible θ values.

Neural Constant Propagation Additionally, the constantness of previous nodes can be propagated throughout the network. $\mathcal{F}$ can integrate this propagation throughout its single traversal. However, $\mathcal{N}$ must take a preliminary traversal to take advantage of the Neural Constant Propagation. This preliminary traversal is $\Theta(n)$ and until parsing the neural network becomes sublinear (which is unlikely), it can improve the computational complexities of $\mathcal{N}$. An algorithmic illustration is given in Algorithm 1.

Example 1. *Assume we have node$_{i,j}$ where*

$$\textit{node}_{i,j} = [w_{i,j,b}, w_{i,j,0}, w_{i,j,1}] = [-1, 5, -1].$$

Naïvely, the Neural Constantness Heuristic would fail since

$$SV_{i,j} = -2 \ngeq 0 \textit{ and } GV_{i,j} = 4 \nleq 0. \tag{7}$$

However, if $o_{i-1,0}$ (the Boolean function of node$_{i-1,0}$) is known to be constant, then the respective out-degree weight of node$_{i-1,0}$ for each node in l_i is aggregated with other constants (such as the bias weight) if $o_{i-1,0}$ is true, else the weight is omitted if $o_{i-1,0}$ is false. So,

$$\textit{node}_{i,j} \textit{ is effectively } [w_{i,j,b} + o_{i-1,0} \times w_{i,j,0}, w_{i,j,1}].$$

Illustrating both cases with the Neural Constantness Heuristic,

If $o_{i-1,0}$ is true, then node$_{i,j}$ $= [-1 + 1 \times 5, -1] = [4, -1]$.

$$SV_{i,j} = 3 \geq 0 \textit{ so } o_{i,j} = 1. \tag{8}$$

If $o_{i-1,0}$ is false, then node$_{i,j}$ $= [-1 + 0 \times 5, -1] = [-1, -1]$.

$$GV_{i,j} = -1 \leq 0 \textit{ so } o_{i,j} = 0. \tag{9}$$

Brute Force With a complexity of $\Theta(2^n n)$, the brute force method (see Algorithm 2) is the simplest and least efficient manifestation of $\mathcal{B}$ (a general node to Boolean function translation algorithm). Since the outputs of the previous layer (out$_{i-1}$) are binarized, the dot product $\theta = \text{out}_{i-1} \cdot \text{node}_{i,j}$ can be considered a bitmask over node$_{i,j}$. Furthermore, this algorithm considers every possibility (every bitmask) in the input space (i.e. the out$_{i-1}$ space). Equivalently, every possible bitmask can be defined by an n-ary Cartesian power

$$\text{inputSpace(node}_{i,j}) = \{0,1\}^n.$$

where $n = \text{length(out}_{i-1}) - 1$. The one is subtracted since the bias output is in $\{1\}$—it is constant.

Algorithm 1: Neural Constant Propagation.

```
Input:
  • node$_{i,j}$: an array of in-degree weights *without* the bias weight.
  • $cm_{i,j}$: an array denoting the constantness of the respective weight in node$_{i,j}$.
  • bias: the bias weight.
Output:
  • $n_{i,j}$: the equivalent of node$_{i,j}$ with constant input nodes considered.
  • bias: the aggregated bias weight
$n_{i,j} \leftarrow \emptyset$;
for $(w_{i,j,k}, m_{i,j,k}) \in (node_{i,j}, cm_{i,j}))$ do
    if $m_{i,j,k}$ is true then
        bias ← bias $+w_{i,j,k}$;
    else if $m_{i,j,k}$ is false then
        continue;
    else
        $n_{i,j} \leftarrow n_{i,j} \cup w_{i,j,k}$;
    return $n_{i,j}$, bias
```

Algorithm 2: Neural node to Boolean function via Brute Force.

```
Input:
  • node$_{i,j}$: a vector of in-degree weights *without* the bias weight.
  • $\theta_{\mathcal{D}}$: the threshold on the domain.
  • bias: the bias weight.
Output: table: a set of every bitmask that yields true for node$_{i,j}$.
table$_{i,j} \leftarrow \emptyset$;
forall mask $\in \{0,1\}^{length(node_{i,j})}$ do
    if node$_{i,j}\cdot$ mask + bias $\geq \theta_{\mathcal{D}}$ then
        table$_{i,j} \leftarrow$ table$_{i,j} \cup$ mask;
return table$_{i,j}$;
```

Algorithm 3: $\mathcal{N}$: Reverse traversal.

```
Input:
  • NN $= [l_1, l_2, \ldots, l_m]$.
  • $\theta_{\mathcal{D}}$: the threshold on the domain.
Output: NNBG: a BG approximation of the neural network.
ct.out = [out$_0$, out$_1, \ldots$, out$_{i-1}, \ldots$ ,out$_{m-1}$] ← reduced $l_i$ input spaces using Neural Constant Propagation;
/* ct.bias$_{i,j}$ is the aggregated bias for node$_{i,j}$ */
ct.bias ← aggregated biases from the Neural Constant Propagation;
/* Each node in the final layer always matters. */
DoCurrentInputsMatter ← [true$_0$, true$_1, \ldots$, true$_{\text{length}(l_m)}$];
for $l_i \leftarrow l_m, l_{m-1}, \ldots, l_1$ do
    inputSpace ← ct.out$_{i-1}$;
    forall node$_{i,j} \in l_i$ do
        if DoCurrentInputsMatter[j] then
            DoPrevInputsMatter, $o_{i,j} \leftarrow$ $\mathcal{B}$(node$_{i,j}, \theta_{\mathcal{D}}$, ct.bias$_{i,j}$, inputSpace);
    LayerBG$_i \leftarrow o_{i,0}|o_{i,1}|\ldots|o_{i,n}$ ;
    if true is not in DoPrevInputsMatter then
        break;
    DoCurrentInputsMatter ← DoPrevInputsMatter;
NNBG ← LayerBG$_i$ >> LayerBG$_{i+1}$ >> $\cdots$ >> LayerBG$_m$;
return NNBG
```

Algorithm 4: $\mathcal{F}$: Forward traversal.

```
Input:
  • $NN = [l_1, l_2, \ldots, l_m]$.
  • $\theta_{\mathcal{D}}$: the threshold on the domain.
Output: NNBG: a BG approximation of the neural network.
for $l_i = l_1, l_2, \ldots, l_m$ do
    inputSpace ← out$_{i-1}$ reduced by considering shared logic and the Neural Constant Propagation;
    bias ← bias updated with the Neural Constant Propagation;
    forall node$_{i,j} \in l_i$ do
        $o_{i,j} \leftarrow \mathcal{B}$(node$_{i,j}, \theta_{\mathcal{D}}$, bias, inputSpace);
    LayerBG$_i \leftarrow o_{i,0}|o_{i,1}|\ldots|o_{i,n}$ ;
NNBG ← LayerBG$_1$ >> LayerBG$_2$ >> $\cdots$ >> LayerBG$_m$;
return NNBG
```

3.3 $\mathcal{N}$ and $\mathcal{F}$

Walkthroughs These walkthroughs use the simplest versions of $\mathcal{B}$: brute force (Algorithm 2) and the Neural Constantness Heuristic (Section 3.2) with Neural Constant Propagation (Section 3.2). Recall that

$$\theta = \text{out}_{i-1} \cdot \text{node}_{i,j}$$

$\mathcal{N}$ **Walkthrough:** Figure 3a is the correct abstraction of Figure 2 via $\mathcal{N}$.

Proof via the Neural Constantness Heuristic. $\mathcal{N}$ begins with the output node, node$_{3,0}$ = $[1.0, 0.1, 1.0]$.

$$\text{SV}_{3,0} = 1 \geq 0$$

so

$$o_{3,0} = 1.$$

□

Proof via brute force. $\mathcal{N}$ begins with the output node, node$_{3,0}$ = $[1.0, 0.1, 1.0]$.

Using $\sigma_{\mathbb{B}}$, the truth table for $o_{3,0}$ is

$o_{2,0}$	$o_{2,1}$	θ	$o_{3,0}$
0	0	1	1
0	1	2	1
1	0	1.1	1
1	1	2.1	1

Since all of the outputs are true, $o_{3,0}$ can be simplified to a constant true. □

$\mathcal{F}$ **Walkthrough:** Figure 3b is the correct abstraction of Figure 2 via $\mathcal{F}$; albeit less interpretable and more computationally expensive than $\mathcal{N}$'s abstraction.

Proof comparing Neural Constantness Heuristic and Brute Force. Begin by enumerating the input layer's input space.

$$\text{out}_0 = n_1 \times n_2 = \{0,1\}^2 = \{[0,0],[0,1],[1,0],[1,1]\} \tag{10}$$

Next, we parse layer one (l_1). Using the Neural Constantness Heuristic for node$_{1,0} = [1,1,1]$,

$$\text{SV}_{1,0} = 1 \geq 0, \text{ so } o_{1,0} = 1. \tag{11}$$

For node$_{1,1} = [-1.5, -1, 2]$,

$o_{0,0}$	$o_{0,1}$	θ	$o_{1,1}$
0	0	-1.5	0
0	1	.05	1
1	0	-2.5	0
1	1	-0.5	0

By the table above,

$$o_{1,1} = \neg o_{0,0} o_{0,1}. \tag{12}$$

Now doing layer 2. Recall that $o_{1,0} = 1$, so out$_1$ is reduced to [1,0] and [1,1]. This is why the bias is aggregated in Section 3.2 and Algorithm 1. Updating node$_{2,0} = [1, -2, 3]$ with the Neural Constant Propagation yields

$$\text{node}_{2,0} \leftarrow [1 + 1 \times -2, 3] = [-1, 3].$$

Then,

$o_{1,1}$	θ	$o_{2,0}$
0	-1	0
1	2	1

Considering the input space without bias aggregation gives an equivalent result. For node$_{2,0} = [1, -2, 3]$,

$o_{1,0}$	$o_{1,1}$	θ	$o_{2,0}$
1	0	-1	0
1	1	2	1

Both methods find that

$$o_{2,0} = o_{1,1}. \tag{13}$$

For node$_{2,1} = [-1, 0.1, 2]$,
node$_{2,1} \leftarrow [-1 + 1 \times 0.1, 2] = [-0.9, 2]$,

$o_{1,1}$	θ	$o_{2,1}$
0	-0.9	0
1	1.1	1

so

$$o_{2,1} = o_{1,1}. \tag{14}$$

For node$_{3,0} = [1, 0.1, 1]$, we use the Neural Constantness Heuristic,

$$\text{SV}_{3,0} = 1 \geq 0, \text{ so } o_{3,0} = 1. \tag{15}$$

□

Output Aggregation Often, the output is not constant and we must aggregate each node's output into a final Boolean expression.

Aggregation can be done by substitution. Using the outputs in the $\mathcal{F}$ walkthrough:

$$o_{3,0} = \neg o_{2,1} \neg o_{2,0} + o_{2,1} o_{2,0} = \neg o_{1,1} \neg o_{1,1} + o_{1,1} o_{1,1}$$

$$= \neg o_{1,1} + o_{1,1} = \neg(\neg o_{0,0} o_{0,1}) + \neg o_{0,0} o_{0,1} = 1.$$

BG Reflection It is straightforward to understand how $\mathcal{N}$ and $\mathcal{F}$ both arrived at $po_0 = 1$ in Figure 3. However, the extra logic found with $\mathcal{F}$ requires further inspection. Notice that $o_{0,1} = \neg o_{0,0} o_{0,1}$ and $o_{0,0} = n_1$ and $o_{0,1} = n_2$. In Figure 3b, node 5 (the 5 is just a label with no logical value) lists all sequences of n_1, n_2 yielding true. So, 01 1 means node 5 equals $\neg n_1 n_2$. The logic for node 5 reflects $o_{0,1} = \neg o_{0,0} o_{0,1}$ which is the Boolean output of node$_{0,1}$.

4 Node-Space and Weight-Space Reductions

As seen in Algorithm 3 and Section 3.3, $\mathcal{N}$ skips negligible neural nodes entirely. Additionally, if an entire layer is negligible, $\mathcal{N}$ ceases parsing layers and finds that the output is constant. The ability to skip the processing of neural nodes is the main advantage of $\mathcal{N}$ over $\mathcal{F}$ (Algorithm 4). However, since $\mathcal{F}$ has already approximated l_{i-1}, l_i can assume a more reduced weight space than $\mathcal{N}$ using shared logic and Neural Constant Propagation. $\mathcal{N}$ only reduces its weight space by performing a preliminary Neural Constant Propagation over the neural network.

4.1 $\mathcal{F}$—Forward Traversal

In contrast to $\mathcal{N}$, $\mathcal{F}$ has a finer understanding of the current layer's input space (l_i) since the previous layer (l_{i-1}) has already been approximated. In addition to knowing if a prior node is constant, $\mathcal{F}$ can make reliable deductions from shared logic among two or more nodes in l_{i-1}.

Reduction by Constants This reduction technique applies to $\mathcal{F}$ and $\mathcal{N}$. Recall that since $o_{1,0}$ was a constant of 1 by the Neural Constantness Heuristic (Equation (11)), l_2's nodes only considered an input space of {[1,0], [1,1]} instead of the exhaustive $\{0,1\}^2$ (Equation (13) and Equation (14)). Furthermore, the Neural Constant Propagation can cause beneficial cascading effects as seen in Example 1.

Shared Logic Assume the brute-force method is applied for node$_{3,0}$ in $\mathcal{F}$. Before simplification, the input space is the exhaustive 2^n space.

$o_{2,0}$	$o_{2,1}$	θ	$o_{3,0}$
0	0	1	1
0	1	2	1
1	0	1.1	1
1	1	2.1	1

$$o_{3,0} = 1$$

However, recall that $o_{2,0} = o_{1,1} = o_{2,1}$ (Equation (13) and Equation (14)). out$_2$, the input space for l_3, is then reduced to {[0,0], [1,1]} since [0,1] and [1,0] will never occur. This realization allows l_3's input space to be reduced by half.

$o_{2,0}$	$o_{2,1}$	θ	$o_{3,0}$
0	0	1	1
1	1	2.1	1

$$o_{3,0} = 1$$

This technique can be extended to greater complexities. For example, if out$_2$ $= [o_{2,0}, o_{2,1}, o_{2,2}]$ and $\mathcal{F}$ deduces that $o_{2,0} = o_{2,1} + o_{2,2}$, then the input space for l_3 can be reduced from 2^3 to 2^2 by removing the four impossible instances: $[1, 0, 0]$, $[0, 1, 1]$, $[0, 1, 0]$, and $[0, 0, 1]$.

A reduced input space does not always yield an equivalent expression as shown above. The input space is reduced by asserting logical statements over primitive inputs (i.e. asserting the expression satisfies some properties such as $o_{2,0} = o_{2,1} + o_{2,2}$). This can only maintain or reduce the number of true outputs in a logic table.

Shared Logic Setbacks The shared logic advantage of $\mathcal{F}$ has three notable setbacks. First, finding the shared logic may exhibit diminishing returns. Second, while reducing the available input space can yield simpler Boolean approximations, post-hoc Boolean simplification can achieve the same result. Third, the logic table may be shortened, but the instances of $\mathcal{B}$ mentioned in this paper do not inherently take advantage of input spaces not in $\{0, 1\}^n$. Future work can solve, mitigate, and/or balance these setbacks.

4.2 $\mathcal{N}$—Reverse Traversal

Node-Space Reduction By establishing which outputs, out$_{i-1}$ were used in each node$_{i,j}$ $\in l_i$, we can determine which nodes in l_{i-1} are negligible and can be subsequently skipped. This is the main advantage of $\mathcal{N}$ over $\mathcal{F}$.

Reduction by Constants Unlike $\mathcal{F}$, $\mathcal{N}$ must take a preliminary sweep to reduce the input-space for each layer. The Neural Constant Propagation of the entire neural network can be done in $\Theta(n)$ time where n is the number of weights. Each node's constantness is then remembered (i.e by a 2D matrix or an associative array). Unless decompositional methods of converting NNs to BGs becomes sublinear (which is unlikely), this initial traversal will improve average complexities for neural networks with at least x percent of constant nodes as defined in the following example.

Example 2. *If out$_{i-1}$ has x percent of constant Boolean expressions, and $\mathcal{B}$ is Chan and Darwhiche's algorithm with a $O(2^{0.5n}n)$ runtime, then the new runtime for each node in l_i becomes $O(2^{0.5t}t)$ where $t = (1 - x)n < n$.*

Use case: if 10% of Boolean expressions in out$_{i-1}$ are constant ($x = 0.1$), then $t = 0.9n$ and the runtime becomes $O(2^{0.5t}t) = O(2^{0.45n}n)$ for each node in l_i.

Shared Logic Detecting shared logic to reduce the input space without approximating the entire node (thus defeating the purpose for $\mathcal{N}$) is left for future work.

5 Conclusion and Future Work

$\mathcal{F}$ versus $\mathcal{N}$ This work contrasts two traversal algorithms: $\mathcal{F}$ (the forward traversal in Algorithm 4) and $\mathcal{N}$ (the reverse traversal in Algorithm 3). Since $\mathcal{F}$ has already approximated the previous neural layer, $\mathcal{F}$ inherently knows which neural nodes are considered constant and can leverage shared logic to minimize the current layer's input space. However, the algorithms to approximate neural nodes mentioned in this paper do not inherently consider an input space that is not in $\{0, 1\}^n$. In contrast, $\mathcal{N}$ omits some neural nodes completely and can have its weight space reduced by doing a preliminary constant propagation sweep in $\Theta(n)$ time. For most use cases, $\mathcal{N}$ is the better choice. Furthermore, the early termination available in $\mathcal{N}$ omits extra computational load and potentially confusing logic found by $\mathcal{F}$ (see Figure 3).

Future work may find ways to apply the shared logic technique in $\mathcal{F}$ to $\mathcal{N}$.

Constantness The Neural Constant Propagation (Section 3.2) finds at least as many constant nodes when compared to the naïve approach (the potential increase is dependent on the network size). However, many neural networks possess no constant nodes as defined in Section 3.2. Nevertheless, the $\Theta(n)$ preliminary sweep is a great tool to predict if algorithms defined by Choi et al. and Briscoe will yield a constant value for the output thus avoiding the exponential algorithm.

This node constantness approach allows for many avenues for future work. One open question is which neural networks and data sets are most susceptible to "constant" neural nodes? Initial investigations seem to point towards binary neural networks with a strong majority class (such as 95% or greater).

Additionally, future work can include a wider threshold for SV and GV (see Section 3.2). For example, if $\text{SV}_{i,j} \geq -0.4$ (and $\text{GV}_{i,j} \leq 0.4$) is checked for $\theta_{\mathcal{D}} = 0$ with the σ activation function, $\text{SV}_{i,j}$ will then consider a node that is guaranteed to be over $\sigma(-.4) \approx 0.4$ as 1 (and $\text{GV}_{i,j}$ under $\sigma(.4) \approx 0.6$ as 0). This marginal loss in accuracy could substantially defray the exponential algorithm $\mathcal{B}$. Furthermore, approximating nodes guaranteed to be within $\sigma(\pm 0.4) \approx (0.4, 0.6)$ as a constant 0.5 (thus aggregated to the bias constant) is more accurate than a binary cast and allows for more constant nodes.

References

Andrews, R.; Diederich, J.; and Tickle, A. 1995. Survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge-Based Systems* 6:373–389.

Baehrens, D.; Schroeter, T.; Harmeling, S.; Kawanabe, M.; Hansen, K.; and Müller, K.-R. 2010. How to explain individual classification decisions. *The Journal of Machine Learning Research* 11:1803–1831.

Briscoe, J. 2021. *Comprehending Neural Networks via Translation to And-Inverter Graphs*.

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

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

Choi, A.; Shi, W.; Shih, A.; and Darwiche, A. 2017. Compiling neural networks into tractable boolean circuits. *intelligence*.

Danks, D., and London, A. J. 2017. Regulating autonomous systems: Beyond standards. *IEEE Intelligent Systems* 32(1):88–91.

Fiesler, E. 1992. Neural network formalization. Technical report, IDIAP.

Guidotti, R.; Monreale, A.; Turini, F.; Pedreschi, D.; and Giannotti, F. 2018. A survey of methods for explaining black box models. *ACM Computing Surveys* 51.

Kingston, J. K. 2016. Artificial intelligence and legal liability. In *International Conference on Innovative Techniques and Applications of Artificial Intelligence*, 269–279. Springer.

Kroll, J. A.; Barocas, S.; Felten, E. W.; Reidenberg, J. R.; Robinson, D. G.; and Yu, H. 2016. Accountable algorithms. *U. Pa. L. Rev.* 165:633.

Ramachandran, P.; Zoph, B.; and Le, Q. V. 2017. Searching for activation functions. *CoRR* abs/1710.05941.

Shi, W.; Shih, A.; Darwiche, A.; and Choi, A. 2020. On tractable representations of binary neural networks.