Reducing the Neural Network Traversal Space for Boolean Abstractions

Jarren Briscoe[1] , **Second Author**[1] , **Third Author**[2,3] , **Robert Ball**[1]
[1]Weber State University
[2]Second Affiliation
[3]Third Affiliation
jarrenbriscoe@gmail.com, {robertball, second}@weber.edu

Abstract

A reverse traversal ($\mathcal{N}$) of the neural network is contrasted with a forward traversal ($\mathcal{F}$). 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 amount of neural nodes approximated.

1 Introduction

1.1 Background and Purpose

Neural networks (NNs) are powerful machine learning techniques, yet they are exceptionally difficult to understand due to their intricate structure. This esoteric nature has inhibited wide-spread adoption of neural networks for lack of accountability (Kroll et al. 2016), liability (Kingston 2016), and safety (Danks and London 2017) for sensitive topics such as predicting qualified job candidates and medical treatments. Furthermore, some legal guidelines 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 due to their lack of feasible formal verifications. This work presents and contrasts two neural-network parsing methods to translate neural networks to more interpretable and formally verifiable Boolean graphs (BGs).

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. Since neural network structures and algorithms are quite diverse, I limit neural networks to the most common type in this paper. Specifically, they are fully interlayered-connected, symmetric, first-order neural networks with some finite amount of layers and nodes and binary input (example: 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 as done in (Choi et al. 2017; Shi et al. 2020). For detailed NN terminology and formal definitions, see (Fiesler 1992).

Explaining Neural Networks From the many techniques attempting to 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 are derived: decompositional, eclectic, and pedagogical.

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

On the other hand, a pedagogical approach learns by an oracle or teacher (global learning). This approach allows underlying machine learning structure(s) to be versatile—it can explain a black-box even if the NN is changed to a support-vector machine (SVM). A simple example of a pedagogical approach 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 omitting inputs negligible to the final product (pedagogical and global).

This paper will introduce and contrast two neural network parsing algorithms, $\mathcal{F}$ is strictly decompositional while $\mathcal{N}$ is partly pedagogical and mostly decompositional (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). Fortunately, there are a few far more efficient methods. The exponentially upper-bound methods below yield higher accuracy than the pseudo-polynomial methods.

Exponential Methods "Reasoning about Bayesian Network Classifiers" has a complexity of $O(2^{0.5n}n)$ (Chan and Darwiche 2012). While Chan and Darwhiche's work was done for Bayesian network classifiers, the analogy to neural node approximation was given by (Choi et al. 2017). Another algorithm to approximate neural nodes is given in "Comprehending Neural Networks via Translation to And-Inverter Graphs" with a complexity of $O(2^n)$ (Briscoe 2021). In this paper, $\mathcal{N}$ improves the upper-bound exponential complexity of Chan and Darwhiche's algorithm as applied to neural networks in (Choi et al. 2017) with no loss in accuracy—assuming a certain percentage (x) of nodes eligible for the Neural Constantness Heuristic.

Pseudo-Polynomial Methods Even quicker methods to compute a neural node with less accuracy includes a pseudo-polynomial $O(n^2W)$[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). Variables were substituted for consistency with this paper.

If one uses a constant maximum depth d for the binary decision tree in Briscoe's work, 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 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)).

$\mathcal{B}$ Generalization To remain concise, I will omit the intricacies to approximate neural nodes in this paper. Instead, I will seek to reduce n (the number of weights) considered with no loss in accuracy. This allows my work to be agnostic of the neural node algorithm which 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 product can be any Boolean structure. Shi et al. and Choi et al. used Ordered Binary Decision Diagrams (OBDDs) and Sentential Decision Diagrams (SDDs) while Briscoe and Brudermueller et al. used And-Inverter Graphs (AIGs) (Shi et al. 2020; Choi et al. 2017; Briscoe 2021; Brudermueller et al. 2020). However, many other Boolean structures exist with different features that are more suitable for different goals (e.g. comprehensibility, Boolean optimizations, and hardware optimizations). I will abstract this final product as BG (Boolean Graph).

1.3 Format

This paper is organized as follows. Section 2 generalizes allowable activation functions and thresholds for $\mathcal{N}$ and $\mathcal{F}$. 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 conclusion and addresses some future work.

[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 $O(n^2W)$.

2 Activation Functions

(Shi et al. 2020; Choi et al. 2017) only considered ReLU and sigmoid functions in their work. However, the properties of activation functions allowed must allow a reasonable Boolean cast (one side is zero and the other positive—Conjunctive Limit Constraint), and for one threshold on the range to best represent a function, one side of the threshold must remain above it while the other must remain below it (diagonal quadrants property). Thus, the class of allowed activation functions is extended to any function that satisfies the conjunctive limit constraint (one infinity limit must go to zero while the other infinity limit is positive) and the diagonal quadrants property.

Conjunctive Limit Constraint:

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

The conjunctive limit constraint allows us to treat one side of the threshold as false (analogous to zero) and the other side as true (analogous to a positive number). Functions which have negative and positive limits (such as $\tanh$) are better defined with an inhibitory/excitatory spectrum. It can be experimentally shown that the accuracy for NN to BG methods as done by (Shi et al. 2020; Choi et al. 2017; Briscoe 2021) decrease for functions like $\tanh$. The Swish activation function (Ramachandran, Zoph, and Le 2017) is in between the inhibitory/excitatory and true/false spectrums since there are negative values and it 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 the at least as accurate ReLU.

Converse of Conjunctive Limit Constraint:

$$\lim_{\theta \to -\infty} f(\theta) > 0 \text{ and } \lim_{\theta \to \infty} f(\theta) = 0 \tag{2}$$

Although unorthodox, activation functions satisfying the Conjunctive Limit Constraint's converse can also be considered (the step function $f_{\mathbb{B}}$ must be flipped respectively).

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 or on the boundaries, centered on the threshold $f_{\mathbb{R}}(\theta_{\mathcal{D}}) = \theta_{\mathcal{R}}$. Consider $\sigma_{\mathbb{R}}(\theta) = \frac{1}{1+e^{-\theta}}$. 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: $\sigma_{\mathbb{R}}(0) = 0.5$. Now draw one horizontal and one vertical line going through $\sigma_{\mathbb{R}}(0) = 0.5$ to create 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 $\theta_{\mathcal{R}}$:

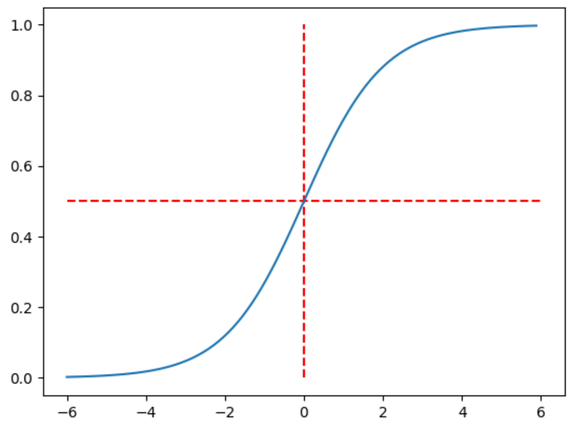


Figure 1: $\sigma(\theta)$ with a threshold of $\sigma(0) = 0.5$ satisfies the diagonal quadrants property.

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

where $\sigma_{\mathbb{B}}(\theta; \sigma_{\mathbb{R}}, \theta_{\mathcal{R}})$ means $\sigma_{\mathbb{B}}(\theta)$ is parameterized by $\sigma_{\mathbb{R}}$ and $\theta_{\mathcal{R}}$.

For monotonic functions, any threshold satisfies the diagonal quadrant property. However, a decent 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 I am limiting this work to activation functions satisfying the diagonal quadrant property, the threshold can be found by only considering the domain. Once the domain's threshold is found, the activation function and threshold in the range can be 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 $(\theta; \theta, 0)$.

$$\text{Henceforth, } \theta_{\mathcal{D}} = 0$$
$$\text{and } \sigma_{\mathbb{B}} = \sigma_{\mathbb{B}}(\theta; \theta, 0)$$

3 Neural Network Traversal

3.1 Final Products

In this section, I introduce the Neural Constantness Heuristic and contrast two network parsing techniques; the forward traversal ($\mathcal{F}$) and the the reverse traversal ($\mathcal{N}$).

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

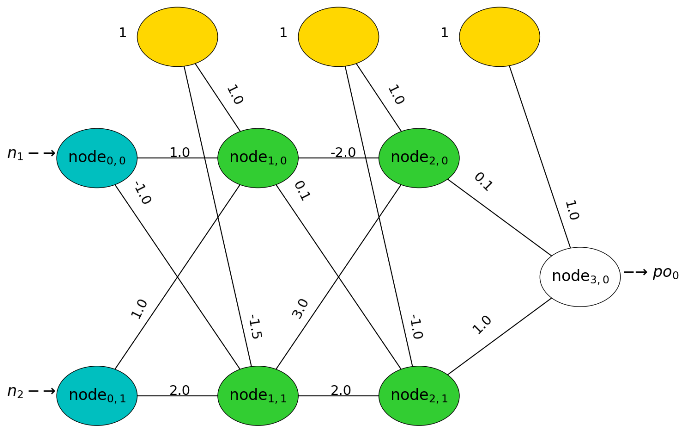
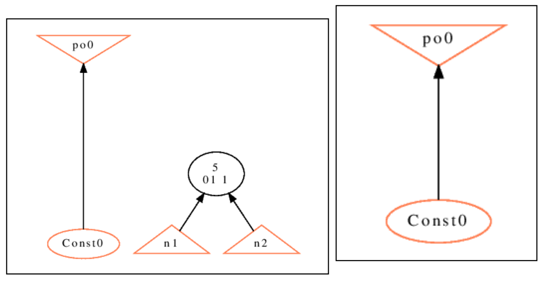


Figure 2: The neural network for the use case presented in Figure 3. $\mathcal{N}$ determines $po_0 = true$ by only parsing the final three weights. The hidden nodes (green) and the output node (white) use the σ activation function.



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

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

Figure 3

When parsing the neural network in Figure 2, $\mathcal{N}$ only had to parse the output node (white) with three weights. In contrast, $\mathcal{F}$ parsed all hidden nodes (green) and the output node for a total of fifteen weights. Additionally, $\mathcal{N}$'s BG was more comprehensible and less prone to machine or human misinterpretation than $\mathcal{F}$'s.

3.2 Algorithms

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

$$NN \leftarrow [l_1, l_2, \ldots, l_m] \tag{3}$$

Each layer, l_i, is an array of nodes.

$$l_i \leftarrow [\text{node}_{i,0}, \text{node}_{i,1}, \ldots, \text{node}_{i,n}] \tag{4}$$

Each node, $\text{node}_{i,j}$, 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 $\text{node}_{i-1,k}$ to $\text{node}_{i,j}$.

$$\text{node}_{i,j} \leftarrow [w_{i,j,b}, w_{i,j,0}, w_{i,j,1}, \ldots, w_{i,j,p}] \tag{5}$$

So NN is a 2D matrix of nodes and a 3D matrix of in-degree weights. In the context of the current $\text{node}_{i,j}$,

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

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

$$\text{out}_i \leftarrow [o_{i,0}, o_{i,1}, \ldots, o_{i,n}] \tag{6}$$

Since the neural network uses the σ activation function, 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:

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

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

In this paper's terminology, the inputs $\vec{x} = \text{out}_{i-1}$, the weights $\vec{w} = \text{node}_{i,j}$, and the bias $b = w_{i,j,b}$.

$$o_{i,j} = \sigma_{\mathbb{B}}(\theta(\text{out}_{i-1}, \text{node}_{i,j} \setminus b, b)) = \begin{cases} 1 & \text{if } \theta \geq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{7}$$

Finally, out_0 consists of binary inputs for the neural network.

Neural Constantness Heuristic The Neural Constantness Heuristic is a heuristic to check for a neural node's constantness with linear computational complexity bounds ($\Theta(n)$). In preparation to understand the heuristic, let us create some variables and functions for conciseness. For $\text{node}_{i,j}$:

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

$$\text{LV}_{i,j} = b + \sum_{w \in W^+} w$$

- Let W^- be the set of weights less than zero in $\text{node}_{i,j} \setminus b$.
- Let $\text{SV}_{i,j}$ be the smallest possible combination of weights in $\text{node}_{i,j}$.

$$\text{SV}_{i,j} = b + \sum_{w \in W^-} w$$

As seen above, the bias weight (b) has a constant input of true so it is always considered. A heuristic to find the binarized output for a neural node is then:

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

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 $\text{node}_{i,j}$. 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 $\text{node}_{i,j}$ 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

$$SV_{i,j} = -2 \ngeq 0 \textit{ and } GV_{i,j} = 4 \nleq 0. \tag{8}$$

However, if $o_{i-1,0}$ (the Boolean function of $\text{node}_{i-1,0}$) is known to be constant, then the respective out-degree weight of $\text{node}_{i-1,0}$ 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 $\text{node}_{i,j} = [-1 + 1 \times 5, -1] = [4, -1]$.

$$SV_{i,j} = 3 \geq 0 \textit{ so } o_{i,j} = 1. \tag{9}$$

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

$$GV_{i,j} = -1 \leq 0 \textit{ so } o_{i,j} = 0. \tag{10}$$

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$_{i,j}$;
```

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 algorithm). Since the outputs of the previous layer (out$_{i-1}$) are binarized, the dot product $\theta =$ out$_{i-1}\cdot$ 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

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

where $n =$ length(out$_{i-1}$) $- 1$. The one is subtracted since the bias output is in $\{1\}$—it is constant. Enumeration over the bias is useless, and including it in the Cartesian power would treat the bias input space as $\{0,1\}$.

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

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

$\mathcal{N}$ Walkthrough: Figure 3b 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

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

□

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

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

Proof. Begin by enumerating the input layer's input space.

$$\text{out}_0 = n_1 \times n_2 = \{0,1\}^2 = \{[0,0],[0,1],[1,0],[1,1]\} \tag{11}$$

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

$$\text{SV}_{1,0} = 3 \geq 0, \text{ so } o_{1,0} = 1. \tag{12}$$

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,

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

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

$$o_{2,0} = o_{1,1}. \tag{14}$$

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

$$o_{2,1} = o_{1,1}. \tag{15}$$

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

$$\text{SV}_{3,0} = 1 \geq 0, \text{ so } o_{3,0} = 1. \tag{16}$$

□

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 $\mathcal{F}$ needs 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 3a, node 5 (the 5 is just a label with no logical value—the other nodes were optimized away in ABC) 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 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 a preliminary Neural Constant Propagation over the neural network.

4.1 $\mathcal{F}$

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 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 (12)), l_2's nodes only considered an input space of {[1,0], [1,1]} instead of the exhaustive $\{0,1\}^2$ (Equation (14) and Equation (15)). Furthermore, the Neural Constant Propagation can cause cascading effects as seen in Example 1.

Shared Logic Assume I instead used the brute-force method 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 (14) and Equation (15)). 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 halved.

$o_{2,0}$	$o_{2,1}$	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). This can only maintain or reduce the amount 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 be a greater cost than it is worth. Second, while reducing the available input space can yield simpler Boolean approximations, post-hoc Boolean simplification can achieve the same result. Second, the logic table may be shortened, but the instances of $\mathcal{B}$ mentioned in this paper do not take inherently take advantage of input spaces not in $\{0,1\}^n$. Future work can solve, mitigate, and/or balance these setbacks.

4.2 $\mathcal{N}$

Node-Space Reduction By saving 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 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.

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 what shared logic can be used 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 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 and potentially confusing logic found by $\mathcal{F}$ (see Figure 3).

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

Constantness The neural constant propagation (Section 3.2) finds strictly more constant nodes as the naïve approach does (the potential increase is dependent on the network size). However, many neural networks have no constant nodes as defined in Section 3.2. Nevertheless, the $\Theta(n)$ preliminary sweep is a great tool to predict if algorithms like (Choi et al. 2017; Briscoe 2021) will yield a constant value for the output; thus preventing the exponential algorithm.

This node constantness idea has 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 LV (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 little loss in accuracy could substantially help 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.

Kingma, D. P., and Ba, J. 2017. Adam: A method for stochastic optimization.

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.