
Making Logic Learnable With Neural Networks

Tobias Brudermueller^{1,2} Dennis L. Shung² Loren Laine² Adrian J. Stanley³ Stig B. Laursen⁴
Harry R. Dalton⁵ Jeffrey Ngu⁶ Michael Schultz⁷ Johannes Stegmaier¹ Smita Krishnaswamy²

Abstract

While neural networks are good at learning unspecified functions from training samples, they cannot be directly implemented in hardware and are often not interpretable or formally verifiable. On the other hand, logic circuits are implementable, verifiable, and interpretable but are not able to learn from training data in a generalizable way. We propose a novel logic learning pipeline that combines the advantages of neural networks and logic circuits. Our pipeline first trains a neural network on a classification task, and then translates this, first to random forests or look-up tables, and then to AND-Inverter logic. We show that our pipeline maintains greater accuracy than naive translations to logic, and minimizes the logic such that it is more interpretable and has decreased hardware cost. We show the utility of our pipeline on a network that is trained on biomedical data from patients presenting with gastrointestinal bleeding with the prediction task of determining if patients need immediate hospital-based intervention. This approach could be applied to patient care to provide risk stratification and guide clinical decision-making.

1. Introduction

Neural networks, equipped with the benefits of differentiable computing, have proven to be effective in *learning* all types of unspecified functions that translate from complex inputs to abstract outputs using training data. However, differentiable computing is not the native language of a com-

puter. All modern computers with the exception of quantum computers are built using digital logic. This discrepancy makes it difficult to implement neural networks directly in hardware. Moreover, the language of logic has a distinct advantage in interpretability and verifiability. When a neural network makes a decision, it is often considered a black box, whereas a chain of logical operations can often be interpreted in terms of the original features, and verified for correct operation in important conditions. On the other hand, there has been very little work in learning logic. Most efforts in logic synthesis focus on minimizing logic that starts out specified. There are no algorithms to truly learn logic from sampled training data such that it generalizes outside the training example. In this work, we seek to combine the learning process of neural networks with the interpretability, verifiability inherent implementability of logic to aid healthcare applications.

We are interested in obtaining a logic representation of a neural network that is trained to make a healthcare decision for several reasons:

- Logical representations can directly be implemented in hardware.
- We can derive an interpretable report consisting of a minimized chain of logical statements leading to a medical decision.
- We can generate new test inputs that would lead to the same decision from the logic representation, using a Boolean SAT solver, for purposes of system verification.

A direct translation of a neural network into an arithmetic circuit uses multipliers, adders and comparators to compute weighted sums and activations. There are two problems with this approach:

- There is a drastic loss of accuracy that is incurred by the necessary quantizing of weights and activations in the arithmetic circuit.
- The arithmetic circuit contains bloated logic, thus, when it is converted into basic gates (such as AND-gates and inverters) contains a large number of logic nodes which can negate interpretability and incur power and area inefficiency in hardware.

¹Institute of Imaging and Computer Vision, RWTH Aachen University, Aachen, Germany ²Yale School of Medicine, New Haven, CT, USA ³Glasgow Royal Infirmary, Glasgow, United Kingdom ⁴Odense University Hospital, Odense, Denmark ⁵Royal Cornwall Hospital, Cornwall, United Kingdom ⁶Christchurch Hospital, Christchurch, New Zealand ⁷Dunedin Hospital, Dunedin, New Zealand. Correspondence to: Smita Krishnaswamy <smita.krishnaswamy@yale.edu>, Tobias Brudermueller <tobias.brudermueller@rwth-aachen.de>.

In this work we show that, surprisingly, the same solution tackles both problems. We show that intermediary translations of the neural network into either a series of cascaded *random forests* or *multi-level look-up tables (LUTs)* before a translation to logic restores accuracy on test data by *adding generalizability*. Indeed, it is often thought that logic models simply *memorize* the data (which is specified in a truth table). To our knowledge, there is only one study (Chatterjee, 2018) that shows certain logic operations such as factoring can lead to generalization. Here we build on this insight and show that several operations on logic in fact generalize. Additionally, we show that these intermediate representations lead to a much smaller logic representation that can be more easily interpreted and converted to hardware.

In order to apply this to decisions in healthcare, we showcase our results on a use case for predicting interventions for patients who come into the hospital with gastrointestinal bleeding. Gastrointestinal (GI) bleeding can be the general symptom of a variety of underlying causes both serious and trivial for a person’s health. Therefore, recent work has gone into training which patients’ GI bleeding symptoms are serious, and therefore require hospitalization. We first train a multilayered neural network on this classification task based on features from the patients hospital record including general information, medical values and medication. Next, we quantize this network’s inputs and activations into multi-bit representations. Then, we present two pipelines that create different final And-Inverter-Graph (AIG) logic representations. AIGs are a flexible representation of logic that can be efficiently transformed into many other commonly used representations without loss of generality (Mishchenko et al., 2006). In one pipeline, we train random forests on each quantitized activation, and then convert the concatenation of these random forests into an AIG. In another pipeline we first train a look-up table network (LogicNet) that we convert to an AIG. Finally the AIG is minimized and rendered into the final logic.

We note that these pipelines are particularly applicable in the biomedical realm. Biomedical diagnoses are based on chains of logical inferences that are performed using lab values, diagnostic criteria and other features. Such decisions cannot be black-box for patient adoption purposes, and Federal Drug Administration (FDA) approval. Rather, we propose to create reports from the AIG structure that lead to the classification decision for particular inputs, along with analysis of critical inputs, which when flipped alter the decision. This allows a doctor to verify the diagnosis and reason about treatments that could flip critical input features. Additionally, the system itself, independent of any features can be verified by generating synthetic input, via a SAT solver like MiniSAT (Eén & Sörensson, 2003). This would lead to positive classification (indicating the need for hospitalization), in order to either gather more data from

specific types of patients or correct errors in the system. Further, there is a need for fast, live diagnosis, which can benefit from hardware implementations. For instance, if a patient is being continuously monitored in a hospital for a specific condition, there is value in having a decision about severity or acuteness decided live using diagnosis on-chip.

2. Background

A *Boolean equation* is a mathematical expression that only uses binary variables. *Logic gates* are simple digital circuits that take a number of binary inputs to produce a binary output, i.e. they perform operations on binary variables and implement Boolean equations. An *AND-Gate* for example outputs 1 when all inputs are 1, while an *OR-Gate* outputs 1 when any input is 1. An *Inverter-Gate* negates the input to form its *complement*. A *truth table* of a gate can store information about which input patterns causes which output. A concatenation of multiple gates can be used to implement a *logic function*. Any Boolean variable or its complement in a Boolean equation is called a *literal*. The AND of one or more literals is known to be a *product* or *cube*. Similarly, the OR of literals is called a *sum*. In this context a *sum-of-products* refers to multiple AND-terms being connected by ORs and vice versa for a *product-of-sums*. Both can be transferred into each other. For a given propositional Boolean formula, the *Boolean satisfiability problem (SAT)* asks whether there exists a satisfying set of variable assignments that makes the output be 1. There are multiple *SAT solvers* publicly available to fulfill this task that is linear in the number of variables (Eén & Sörensson, 2003; Moskewicz et al., 2001; Goldberg & Novikov, 2007).

Alternatively, a Boolean function can be represented by a directed acyclic graph (DAG) in which each node models a logic gate and each edge models a gate connection. A specific implementation of a Boolean function as a DAG is the so-called *And-Inverter-Graph (AIG)*, for which both AND- and Inverter-gates are part of a universal gate set. In this context, a powerful tool that we use is *ABC* (Brayton & Mishchenko, 2010), which provides powerful transformations, such as redundancy removal that leads to a reduced number of nodes and levels. For instance, “when AIG rewriting is performed in ABC, a minimal representation is found among all decompositions of all structural cuts in the AIG, while global logic sharing is captured using a structural hashing table” (Brayton & Mishchenko, 2010). In ABC the gates are also factored such that they only receive two inputs and that the nodes follow a topological order. The *area or size* of an AIG is given by the number of the nodes in the graph, while the *depth or delay* is given by the number of nodes on the longest path of the primary inputs and outputs.

A *look-up table (LUT)* is a table that saves previously calculated results or information in form of array-like entries

that are easily accessible. In Boolean logic, an N -bit LUT can encode a Boolean function with N inputs by storing the corresponding truth table. Thus, in the case of N bits it has 2^N rows, one row for each possible bit-pattern. In digital logic, a LUT is implemented with a *multiplexer* (*MUX*), which has select lines. Those are driven by the inputs to the Boolean function to access the value of the corresponding output that is stored in the array.

A method which we use in this paper, to which it can be referred as *LogicNet*, is formed by a combination of look-up tables (Chatterjee, 2018). Those LUTs are arranged in successive layers, similar to how it is done in neural networks. But a key difference is that the training process does not involve a backpropagation and instead rather is a memorization process. Each LUT in a layer receives inputs from only a few LUTs in the previous layer, for which the connections are chosen at random. The number of outcome columns of a single LUT depends on the number of different outputs that can be observed for the input patterns. Each entry in a row in the outcome columns counts how many times the pattern is associated with the outcome in the given data set. Hence, LogicNet is a network of concatenated look-up tables with multiple layers and serves the memorization of the information given by the data, but also includes noise.

Unlike multiplexers, other arithmetic circuit pieces exist to fulfill the task of adding two Boolean numbers, multiplying them or comparing them against each other. The components to realize such calculations are called *adder*, *multipplier*, and *comparator* respectively. It is common practice in industry and academia to describe logic circuits high-level with so-called *hardware description languages*, which basically are programming languages to define input and output declarations, operations and timing. Additionally, they allow the circuits to be simulated on a set of Boolean input vectors.

Decision trees are a popular method in machine learning, due to their interpretability (Hara & Hayashi, 2016). They are particularly suitable for classification tasks where the feature space can be separated into distinct bins which can be shown as a tree-structure. When using multiple trees in an ensemble and letting them vote on the classification, this substantially improves the accuracy, which leads to a method known as *random forests*. The growing of a random forest is usually based on random vectors that govern the growth of each tree in an ensemble (Breiman, 2001).

In this work we translate trained neural networks into AND-Inverter-Graphs (AIG) using three different processes. They use different operations for logic synthesis and therefore result in different logical minima, which we compare to assess generalization and gate count. Our work only covers *combinational logic*, which means that the output of the logic is only dependent on the inputs, but not on time. The

three pipelines can be described as follows:

1. A trained neural network is turned into an arithmetic circuit and then compiled into an AIG.
2. Sets of activations from a trained neural network are quantized to form multiple training data sets. Subsequently, multiple random forests are trained on them and turned into one circuit of muxes and comparators before a translation to AIG.
3. Similarly, LogicNets are trained on quantized activations of a trained neural network. Those are again used to form a circuit of muxes and comparators before the compilation into AIG.

3. Related Work

Machine Learning and Logic: The intersection of machine learning and logic has been exploited in Chatterjee & Mishchenko (2019), which showed that circuit-based simulations can be used as an intrinsic method of detecting overfitting of a machine learning model. Contrastingly, Chatterjee (2018) investigated the trade-off between memorization and learning with LogicNet. Other work focused on the relationship of neural networks and Boolean satisfiability. For instance, Bünz & Lamm (2017) and Selsam et al. (2018) use deep neural networks to learn solving satisfiability problems as an alternative to SAT-solvers. Contrastingly, Xu et al. (2018) and Prasad et al. (2007) make use of deep learning to solve problems related to binary decision diagrams, such as variable ordering or estimation of state complexity.

Logic Synthesis and Hardware Implementations: There is a substantial amount of work about implementing already trained neural network architectures on Field Programmable Gate Arrays (FPGAs), such as the frameworks provided in Venieris & Bouganis (2016), which employs a synchronous dataflow model of computation, Jung & su Kim (2007) which allows real-time control of the backpropagation learning algorithm or Wang et al. (2019) where the FPGAs' native LUTs are used as inference operators. In a regular setting of neural networks being compiled into hardware (McDanel et al., 2017; Zhao et al., 2017; Fraser et al., 2017; Umuroglu et al., 2017) the starting point are binarized neural networks (Hubara et al., 2016), which have binarized weights and activations during training and thus have a different training behaviour than standard neural networks. To the best of our knowledge, there is no work to learn logic gate structures by using quantized neural network activations, and compiling them into And-Inverter-Graphs.

Advances in Health Care: According to Topol (2019), the advances of deep learning directly affect the field of medicine and health care at all three levels: clinicians, health

systems, and patients. Especially, for many diseases, accurate and timely clinical decision making is critically required. Lengthy and costly procedures are, however, a bottleneck at this point and can potentially be addressed by low-cost diagnostic tools that run on chips, close to where sensors produce data. Hence, there is a high demand for miniaturization and so-called *lab-on-chip (LoC) technology* (Wu et al., 2018). We address this issue in our work by providing a novel pipeline to translate neural networks into logic gates which can serve as further input to industrial logic synthesizers to produce on-chip machine learning designs.

Interpretability: In their work, Murdoch et al. (2019) define *interpretability* in the context of machine learning as “the use of machine-learning models for the extraction of relevant knowledge about domain relationships contained in data.” By means of this, relevance refers to knowledge and insights that are needed by an audience in a specific domain problem. For that reason, in some cases interpretable models that are less accurate than non-interpretable ones might even be preferred (Ribeiro et al., 2016). In Du et al. (2019), it is distinguished between *intrinsic* and *post-hoc interpretability*. The first refers to self-explanatory models due to their structures, which for example includes decision trees and other rule-based learning methods. The latter involves deriving a second model for explaining the first one, which our work of deriving random forests, look-up tables and logic from neural networks belongs to. Our novel pipeline is also inspired by Lundberg et al., where gradient-boosted decision trees that provide local explanations are combined to represent global structures of a model. They show that especially on medical data this can leverage rich summaries of both an entire model and individual features.

4. Methods

4.1. Logic Operations and Generalization

Here, we show that general logic synthesis involves memorizing specific inputs and thus does not generalize to unseen inputs. This motivates us to start from a trained neural network and then learn logical representations. Next, we show that don’t care minimization, offers a K-nearest-neighbor-like interpolation based on cube expansion, thus motivating our use of random forests as an intermediate step for improved accuracy as well as minimized hardware complexity. Finally, we show that factored forms like multi-levels of LUTs can also induce generalization to improve accuracy as well as minimize logic area.

The most direct way to obtain a logic circuit is as a *sum-of-products (SOP)* which is a 2-level logic obtainable directly from the truth-table. Given an n -input logic function $f(x_1, \dots, x_n)$ the circuit consists of an AND gate AND_i for each combination X_i such that $f(X_i)$ is 1, and there is a single OR gate that takes as input each of the AND gates,

$$f(x_1, \dots, x_n) = OR(AND_1, \dots, AND_k).$$

Note that this basic form of logic is exactly memorization in that each training vector that produces a 1 output is encoded directly as an AND gate, thus this function does not generalize outside the training data. This circuit has high complexity in that an n -input AND gate requires a tree of 2-input AND gates for hardware implementation. Thus, an n -input AND gate will require roughly n 2-input AND gates. Generally, hardware synthesis involves a restricted gate library such as K -input ANDs, ORs, NOTs, etc.

The primary goal of logic synthesis, involves reducing the *area* of the circuit corresponding to f such that there are fewer basic gates involved in the building of the circuit. One basic operation in logic synthesis is dependency elimination. In an SOP setting an example of dependency elimination is cube expansion $ab + a\bar{b} = a$. Here, the input b is eliminated due to lack of dependency on its value since both 0 and 1 values of b can evaluate to the same output solely depending on the value of a . We assert that dependency elimination in the case of unseen inputs, which we term *don’t care-based dependency elimination*, can lead to generalization.

Example: Here, for an n -input logic function $f(x_1, \dots, x_n)$ we define *don’t care-based single-variable elimination* as setting $f(x_1 \dots, x_{i-1}, 0, x_{i+1} \dots, x_n) = f(x_1 \dots, x_{i-1}, 1, x_{i+1} \dots, x_n)$ if both $x_i = 1$ and $x_i = 0$ are not seen in combination with the other literals.

Suppose we have a set of training samples $\mathcal{X} = \{X_1, \dots, X_m\}$, where $X_i = \{x_{i,1}, x_{i,2}, \dots, x_{i,n}\}$. Suppose the test data $\mathcal{T} = \{T_1, \dots, T_q\}$ is distributed around the training data and generated by a bit flip on any bit of the training data.

Then we start with two SOP circuits, circuit C_1 on the combinations X_i such that $F(X_i) = 1$ and C_2 on combinations X_j such that $F(X_j) = 0$. If $C_1(T_i) == 1$ then we assign a 1 as the response to test input T_i , if $C_2(T_i) == 1$ then we assign response 0, otherwise we pick uniformly at random between 0 and 1.

Now suppose we can create a new circuit C'_1 that randomly eliminates one input from each AND gate in C_1 such that some of the examples at a 1-bit hamming distance from the training data receive the right answer. Suppose we do the same for C_2 to create C'_2 . Now for a test input T_i suppose that we assign T_i to 1 if $C'_1(T_i) \& C'_2(T_i)$ and to 0 if $\bar{C}'_1(T_i) \& \bar{C}'_2(T_i)$ and pick randomly otherwise. Then this means that the test example gets assigned the same value as close training examples and if there is smoothness in the classification decision of the neighboring points, the circuit generalizes.

Based on the intuition provided in this setting, we use a general form of dependency elimination by training a random

forest on each activation of a neural network. Note that the random forest explicitly selects input variables on which to decide the logic function at the activation of the output, and eliminates all other dependencies. Thus, we reason that deriving the logic from first translating a neural network into a series of random forests can improve accuracy on test data.

Another logic operation that has been shown to generalize is *factorization*, which involves adding depth in order to pull out a common term in logic. An example of factorizing is $abc + abd = ab(c + d)$, where the factor ab is pulled out of both terms of the sum. Again cube-expanding this expression to ab would allow an inductive generalization argument where based on ab being the decisive factor in some examples we generalize to others. Chatterjee (2018) showed that factorized LUTs also generalize over simple LUTs, based on pulling common factors out of the training data. This motivates another pipeline we examine where we translate a neural network first to a high-depth LUT network called a *LogicNet* and then to an AIG. Our results show that both pipelines lead to lower complexity (AIG gate count) as well as improved accuracy over the direct translation.

4.2. Logic Interpretation and Verifiability

Logic has additional advantages, particularly in the health care realm. First, if a diagnosis or treatment decision is decided by a logic circuit then a *report* can be created such that the decision can be explained by conjunctions and disjunctions of original factors. Second, using logic allows for the use of SAT solvers, whose speed and performance have been the key factors for success in modern logic synthesis. For a boolean function $f(x)$ a SAT solver, such as MiniSAT in ABC (Brayton & Mishchenko, 2010; Eén & Sörensson, 2003), can find an input vector y such that $f(y) = 1$. The input vector y thus, satisfies the logic result. Such vectors can be generated en masse to test features of the system and potentially add features or change the data set design if the decision is not medically valid. Further, in any such input vector one can analyze the controlling inputs, those whose bit flip causes the decision to change, thus determining the critical and potentially causative factors in the logic. In the appendix (Section 6) we provide run-times of MiniSAT on AIGs of some final logic.

4.3. The Framework

Figure 1 gives an overview of our method that consists of the following steps.

- We train a fully-connected neural network on a classification task with real valued inputs x_i and labels y_i of a training data set \mathcal{X}_t . At each layer l with N_l nodes we have a set of real valued activations $\mathcal{A} = \{A_{l;1}, A_{l;2} \dots A_{l;N_l}\}$ and weights $\mathcal{W} =$

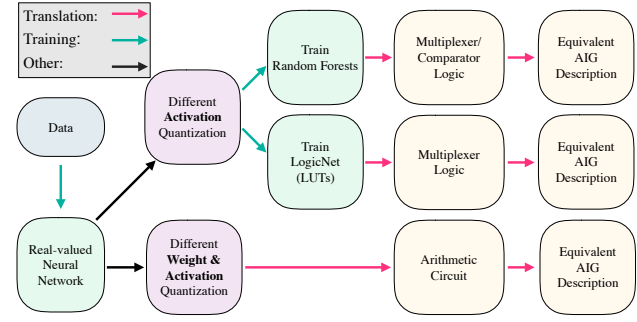


Figure 1. Overview of the proposed pipeline.

$$\{W_{l;1}, W_{l;2} \dots W_{l;N_l}\}.$$

- At each node n of each layer l we quantize each real valued activation $A_{l;n}$ and weight matrix $W_{l;n}$ according to a quantization scheme into m bits and gain $A_{l;n}^q$ and $W_{l;n}^q$ accordingly (see Section 4.3.1; see Figure 1a).
- We translate the neural network into an equivalent *arithmetic circuit* representation by direct conversion of each node's weighted sum into a multiplier and adder circuit. This gives us the logic representation $Logic_{NN}$ (see Section 4.3.2).
- Let $l = 0$ be the input layer and $l = L$ be the output layer of the neural network. For each node $0 \leq n < N_l$ in layer $0 < l < L$, we train a random forest $RF_{l;n}$ and a LogicNet $LGN_{l;n}$ on a data set $Z_{l;n}$ that consists of quantized activations $\{A_{[l-1];k}^q \mid \forall k \in [0, N_{l-1}]\}$ from the previous layer as features and the corresponding quantized activation $A_{l;n}^q$ from node $l; n$ as label (see Figure 1b).
- We now translate each $RF_{l;n}$ and $LGN_{l;n}$ into equivalent logic $RF_{l;n}^{logic}$ and $LGN_{l;n}^{logic}$ (see Section 4.3.3 and 4.3.4) and refer to this as *module* or interchangeably as *block* (see Figure 1b).
- For each of the methods, random forest and LogicNet, we take each logic module and cascade them together to reform the entire neural network structure, which yields the full logic $Logic_{CRF}$ and $Logic_{CLGN}$ (see Figure 1c).
- Lastly, for each $j \in \{NN, RF, LGN\}$ we create an And-Inverter-Graph AIG_j from $Logic_j$ using the ABC tool (Brayton & Mishchenko, 2010) (see Figure 1d).

4.3.1. QUANTIZATION SCHEME

The neural network's weights, activations, and training data, are typically floating point values that have a limited precision of 32 or 64 bits. We want to derive binary strings

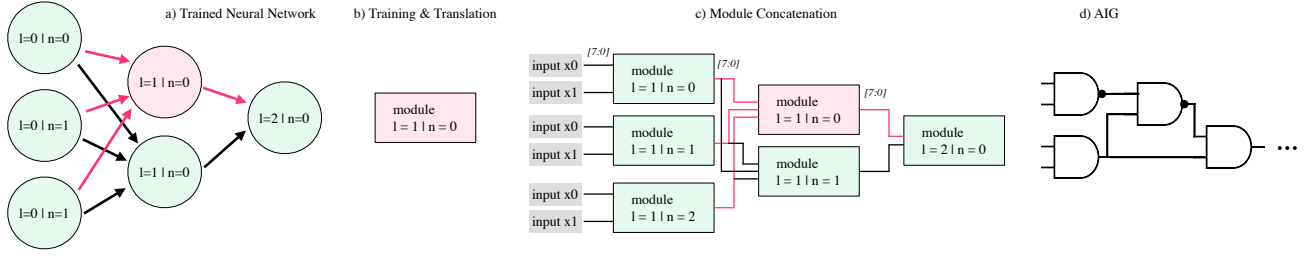


Figure 2. Illustration of translation from neural networks to logic (a) trained neural network (b) intermediate input-output module extracted for training a LogicNet or random forest module (c) concatenation of modules to recreate entire network (d) translation of LogicNet/random forest to AIG.

from these floats with a lower precision of m bits, while maintaining information from before and after the decimal point and about the sign, which needs to be done by binning and clipping in combination with a two-complement binarization. Therefore, we define the number of total bits for the quantization as $m = k + i$ with k being the number of bits that represent the information before the decimal point (*integer bits*) and i for the positions after the decimal point (*fractional bits*). We use this quantization scheme consistently for every value that needs to be quantized. The pseudo-code can be found in Algorithm 1.

Algorithm 1 Quantization Scheme

Input: float value x , number of total bits m , number of fractional bits i

Conversion to Integer-Representation:

$$x_{int} = \text{int}(1 \ll i) * x$$

$$\text{largest_signed_int} = (1 \ll (m - 1)) - 1$$

$$x_{int} = \min(\text{largest_signed_int}, x_{int})$$

$$\text{smallest_signed_int} = -(1 \ll (m - 1))$$

$$x_{int} = \max(\text{smallest_signed_int}, x_{int})$$

$$\text{largest_unsigned_int} = (1 \ll m) - 1$$

$$x_{int} = x_{int} \& \text{largest_unsigned_int}$$

Conversion to Binary String:

return $\text{format}(x_{int}, 'b').zfill(m)$

4.3.2. TRANSLATION TO AN ARITHMETIC CIRCUIT

Figure 3 shows how the operations during a forward pass at a neural network’s node can be translated into logic. The multiplications $w_i * x_i$ are executed by *multipliers*. The two terms of the multiplication must have the same number of bits $m = k + i$. Due to the nature of binary multiplications, the result is of size $2m$ bits. The summation $\sum_i (w_i * x_i)$ is done by the *accumulator logic*. To prevent an overflow, we assume it to be of size $3m$ bits. Then, a ReLU-activation function is modeled as *comparator logic* with the $input_1$ of the accumulator and a constant $input_2$ of zero, both of size $3m$ bits. The comparator drives a *multiplexer logic*

that outputs $input_1$ if the comparator outputs 1 (given by $input_1 > input_2$) and $input_2$ (meaning zero) otherwise. Finally the result has to be brought back to size m . This is done by a *logical right shift* of $2i$ bits (to remove the additional fractional bits), followed by a clipping to the largest signed integer number and smallest signed integer number that can be represented with m bits and hence, follows the procedure of the quantization scheme. After that, the result is still of size $3m - 2i$, but the relevant information stands at the m least significant bits.

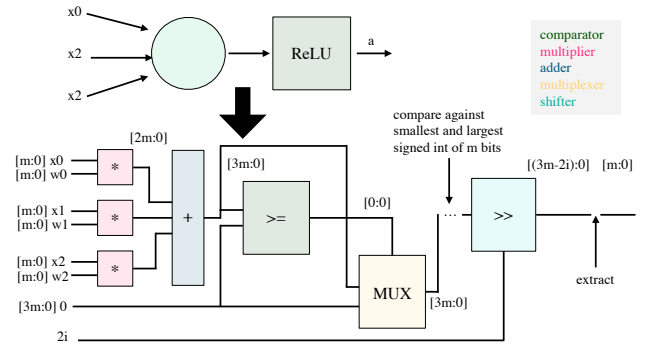


Figure 3. Overview of the logic of a single neuron with a ReLU activation function for a quantization scheme of m bits. Multipliers (with $2m$ bit inputs) are used for the weight-input multiplications and an accumulator (with $3m$ bit capacity) is used for the addition. A comparator is used to compare against a constant input of 0 to model the ReLU function and a shifter is used for a right shift to eliminate the additional i fractional bits before extracting the m least significant bits.

4.3.3. TRANSLATION TO LOGIC: RANDOM FOREST

To be precise, we train a random forest on each bit $0 \leq j < m$ separately and frame it as a binary classification problem. We also turn each random forest separately into logic. Note that subsets of the quantized activation data have to be formed that always extract the j^{th} bit of each feature and label. This procedure is also done for simplifying the logic itself. When only using binary data, the random forest

problem is simplified to thresholds of 0.5 as there are only two possible values 0 and 1. What remains to be learned by the random forests is which feature is relevant for the classification. This is equivalent to learning a removal of “unimportant” variables and hence, to a dependency elimination that can drive generalization as described in Section 4.1. The depth of a decision tree influences the number of extracted features. Note that the final leaves of the forests are class probabilities, which means that an argmax needs to be applied for the actual binary class prediction. The translation of such a single-bit random forest to logic finally leads to a *cascade of comparators and multiplexers*, as shown in Figure 4. To form the final logic module $RF_{l;n}^{logic}$ that models the node $l; n$ we concatenate each of the single bit random forests. Hence, the output of the module is m bits and serves as input to the next module.

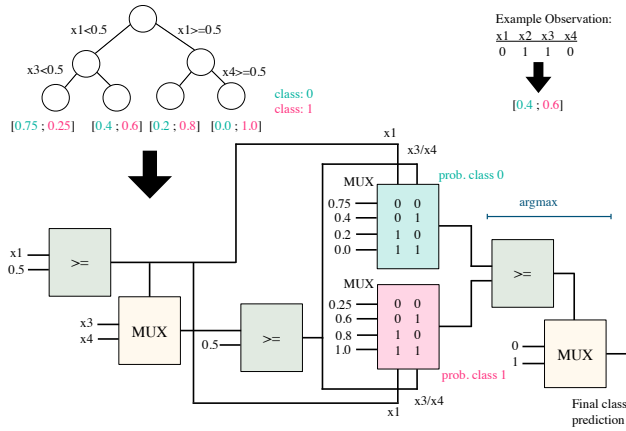


Figure 4. Overview of the logic of a single decision tree of depth 2 for 4 data features with class predictions at the final leaves. It is implemented as a cascade of comparators and multiplexers.

4.3.4. TRANSLATION TO LOGIC: LOGICNET

Similar to the training of random forests, we train LogicNets on binary classifications, i.e. one LogicNet per bit $0 \leq j < m$ which are translated into logic separately. As LogicNet only consists of look-up-tables, the simplest way of implementation is with a cascade of multiplexers, as visualized in Figure 5. The logic of each LogicNet to model node $l; n$ is concatenated to one module to form $LGNet_{l;n}^{logic}$ with N_{l-1} inputs of m bits and an output of m bits.

5. Evaluation

5.1. Data

We use balanced data without missing values from patients presenting with *gastrointestinal (GI) bleeding* at multiple hospitals around the globe (Stanley et al., 2017). Upper

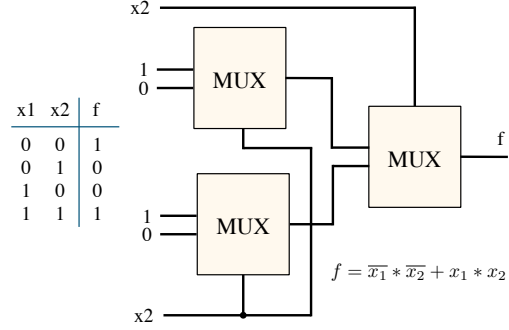


Figure 5. Example of a look-up table (LUT) logic implementation with multiplexers for a given Boolean function. Such LUTs are stacked in multiple layers to form LogicNet.

gastrointestinal bleeding is a common cause of admission to hospital worldwide and can lead to severe outcome, even mortality, when not diagnosed correctly. The data set contains 3,012 observations with 27 numerical features for each observation. The features cover general information about the patient, such as sex and age, medical values, such as systolic blood pressure and the haemoglobin level, and even include previous medication. We use doctors’ decisions for a need of immediate hospital-based intervention as the outcome variable for the binary classification task.

5.2. Measuring Accuracy

To measure test accuracy, i.e., the performance of our model, we use the percentage of correct predictions from the simulation of the logic circuits on the test data set. We also report the number of gates (given by the number of nodes in the AIG) and the depth of the logic (given by the levels of the AIG) as measures of logic complexity.

Note that the different logic circuits under comparison are translations of a single neural network with the same architecture as used in previous work on a broad evaluation of machine learning models on the same data set (Shung et al., 2020). It has one hidden layer with 20 nodes and ReLU activation functions and was trained until convergence with Adam optimizer (Kingma & Ba, 2014). We use a softmax activation on the last layer. The test data set was the same among all experiments.

For all circuits, we examine the influence of the number of total bits and the number of fractional bits in the quantization scheme (see Section 4.3.1). For LogicNet specifically, we investigate different depths (i.e. the number of layers in each LogicNet module) $\{2, 3, 4\}$, widths (i.e. the number of LUTs in each layer) $\{50, 100, 200\}$ and LUT-sizes $\{4, 6, 8\}$. For random forests, we examine different maximal tree depths $\{5, 10, 15\}$ of each logic module and number of

estimators, i.e. decision trees, $\{2, 3, 4\}$.

5.3. Results

As expected, the neural network with real-valued weights achieves the highest accuracy of 82.67% on the test data set and serves as our baseline model. We first test the direct translation of the neural network into an arithmetic circuit $Logic_{NN}$ and into AIG as a result of applying different quantization schemes (as described in Section 4.3). Table 1 provides the results of the logic for different numbers of total bits and fractional bits. The best logic model in this setting achieves 54.17% accuracy for 16 fractional bits out of 32 total bits. However, the AIG has 2.5 million nodes. Table 2 shows the results for this quantization scheme (32 total bits, 16 fractional bits). Our LogicNet approach achieves an accuracy of 64.33% using the same quantization scheme with only 1.3 million nodes and the random forest achieves 55.67% accuracy with 1.9 million nodes in the AIG. Thus, both the LogicNet and random forest approaches reduce hardware complexity *and* display increased accuracy over the direct translation into logic.

Next, we investigate a quantization scheme (8 total bits, 6 fractional bits) that leads to smaller logic, as this is useful for interpretability. Here the direct translation has only $\approx 265K$ nodes but its accuracy is worse than random guessing, at only 46.83%. Thus, the direct translation of logic indeed suffers from *both* drastic loss of accuracy as well as high logic complexity. The LogicNet approach achieves an accuracy of 58.83% in this setting and has only 256K gates in the AIG. While, the random forest has 55.00% accuracy and roughly 267K gates in the AIG. Thus, the LogicNet achieves significantly higher accuracy, while the random forest displays increased accuracy with similar hardware cost. The complete table of results for different settings of the parameters can be found in the supplementary material.

We also provide runtime results of running MiniSAT (Eén & Sörensson, 2003) in the appendix (see Section 6). On logic of these sizes MiniSAT runs in the area of one to a few hours.

Table 1. Results of the **baseline neural network with 82.67% accuracy** (with details as described in Section 6.1) turned into logic with **different quantization schemes**. The column-wise “winner” is marked in bold.

TOTAL BITS	FRACTIONAL BITS	AIG NODES	AIG LEVELS	LOGIC ACCURACY (%)
8	6	264,923	464	46.83
16	6	599,987	704	53.17
16	8	706,548	705	54.50
8	4	210,457	477	51.50
32	16	2,516,251	1,113	54.17

Table 2. Results of the logic from a direct translation of the baseline neural network from Table 1 with a **quantization scheme of 32 total bits and 16 fractional bits in comparison** to the best LogicNet (LGN) and random forest (RF) logic (derived under the same quantization scheme), including their settings. The column-wise “winner” is marked in bold.

LOGIC TRANSLATION	SETTINGS	AIG NODES	AIG LEVELS	LOGIC ACCURACY (%)
DIRECT	-	2,516,251	1,113	54.17
LGN	DEPTH: 4 WIDTH: 50 LUT-SIZE: 4	1,286,186	454	64.33
RF	ESTIMATORS: 3 MAX. DEPTH: 5	1,865,767	519	55.67

Table 3. Results of the logic from a direct translation of the baseline neural network from Table 1 with a **quantization scheme of 8 total bits and 6 fractional bits in comparison** to the best LogicNet (LGN) and random forest (RF) logic (derived under the same quantization scheme), including their settings. The column-wise “winner” is marked in bold.

LOGIC TRANSLATION	SETTINGS	AIG NODES	AIG LEVELS	LOGIC ACCURACY (%)
DIRECT	-	264,923	464	46.83
LGN	DEPTH: 4 WIDTH: 200 LUT-SIZE: 6	255,886	205	58.83
RF	ESTIMATORS: 4 MAX. DEPTH: 5	266,826	252	55.00

6. Conclusion

Here, we presented two novel frameworks for translating trained neural networks into logic gate representations. The motivation for these frameworks was to combine the learnability of neural networks with the interpretability, verifiability and implementability of logic. We noted that, in general, a straightforward translation of a neural network into AND-inverter logic incurs a drastic loss of accuracy. Surprisingly, we found that the inclusion of training random forests or lookup-tables (LUTs) for each activation before the translation to logic increases accuracy and decreases hardware complexity. Thus, these intermediate optimizations on the logic allow for greater generalization than direct translation. Our work is one of the first to provide evidence of the ability of logic operations to generalize to unseen training examples. Future work remains in investigating the properties of our intermediate operations and different quantization schemes.

Acknowledgements

We thank Sat Chatterjee (Google AI, Mountain View, CA, USA), Alan Mishchenko (Department of EECS, University of California, Berkeley, CA, USA), and Claudionor N. Coelho (Google AI, Mountain View, CA, USA) for the discussion and the collaboration on the technical implementations. We thank the following authors for the data collection: Dennis L. Shung (*Yale School of Medicine, New Haven, CT, USA*), Loren Laine (*Yale School of Medicine, New Haven, CT, USA*), Adrian J. Stanley (*Glasgow Royal Infirmary, Glasgow, United Kingdom*), Stig B. Laursen (*Odense University Hospital, Odense, Denmark*), Harry R. Dalton (*Royal Cornwall Hospital, Cornwall, United Kingdom*), Jeffrey Ngu (*Christchurch Hospital, Christchurch, New Zealand*) and Michael Schultz (*Dunedin Hospital, Dunedin, New Zealand*).

References

- Brayton, R. and Mishchenko, A. Abc: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pp. 24–40. Springer, 2010.
- Breiman, L. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001. ISSN 1573-0565. doi: 10.1023/A:1010933404324. URL <https://doi.org/10.1023/A:1010933404324>.
- Bünz, B. and Lamm, M. Graph neural networks and boolean satisfiability. *arXiv preprint arXiv:1702.03592*, 2017.
- Chatterjee, S. Learning and memorization. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 755–763, Stockholmsmssan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL <http://proceedings.mlr.press/v80/chatterjee18a.html>.
- Chatterjee, S. and Mishchenko, A. Circuit-based intrinsic methods to detect overfitting. *arXiv preprint arXiv:1907.01991*, 2019.
- Du, M., Liu, N., and Hu, X. Techniques for interpretable machine learning. *Communications of the ACM*, 63(1): 68–77, 2019.
- Eén, N. and Sörensson, N. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pp. 502–518. Springer, 2003.
- Fraser, N. J., Umuroglu, Y., Gambardella, G., Blott, M., Leong, P., Jahre, M., and Vissers, K. Scaling binarized neural networks on reconfigurable logic. In *Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, pp. 25–30, 2017.
- Goldberg, E. and Novikov, Y. Berkmin: A fast and robust sat-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007.
- Hara, S. and Hayashi, K. Making tree ensembles interpretable. *arXiv preprint arXiv:1606.05390*, 2016.
- Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks. In *Advances in neural information processing systems*, pp. 4107–4115, 2016.
- Jung, S. and su Kim, S. Hardware implementation of a real-time neural network controller with a dsp and an fpga for nonlinear systems. *IEEE Transactions on Industrial Electronics*, 54(1):265–271, 2007.
- Kingma, D. and Ba, J. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.
- Lundberg, S. M., Erion, G., Chen, H., DeGrave, A., Prutkin, J. M., Nair, B., Katz, R., Himmelfarb, J., Bansal, N., and Lee, S.-I. From local explanations to global understanding with explainable ai for trees.
- McDanel, B., Teerapittayanon, S., and Kung, H. Embedded binarized neural networks. In *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks*, EWSN 2019;17, pp. 168173, USA, 2017. Junction Publishing. ISBN 9780994988614.
- Mishchenko, A., Zhang, J. S., Sinha, S., Burch, J. R., Brayton, R., and Chrzanowska-Jeske, M. Using simulation and satisfiability to compute flexibilities in boolean networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(5):743–755, 2006.
- Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pp. 530–535, 2001.
- Murdoch, W. J., Singh, C., Kumbier, K., Abbasi-Asl, R., and Yu, B. Interpretable machine learning: definitions, methods, and applications. *arXiv preprint arXiv:1901.04592*, 2019.
- Prasad, P. C., Assi, A., and Beg, A. Binary decision diagrams and neural networks. *The Journal of Supercomputing*, 39(3):301–320, 2007.

- Ribeiro, M. T., Singh, S., and Guestrin, C. Model-agnostic interpretability of machine learning. *arXiv preprint arXiv:1606.05386*, 2016.
- Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., and Dill, D. L. Learning a sat solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*, 2018.
- Shung, D. L., Au, B., Taylor, R. A., Tay, J. K., Laursen, S. B., Stanley, A. J., Dalton, H. R., Ngu, J., Schultz, M., and Laine, L. Validation of a machine learning model that outperforms clinical risk scoring systems for upper gastrointestinal bleeding. *Gastroenterology*, 158(1):160–167, 2020.
- Stanley, A. J., Laine, L., Dalton, H. R., Ngu, J. H., Schultz, M., Abazi, R., Zakko, L., Thornton, S., Wilkinson, K., Khor, C. J., et al. Comparison of risk scoring systems for patients presenting with upper gastrointestinal bleeding: international multicentre prospective study. *bmj*, 356, 2017.
- Topol, E. J. High-performance medicine: the convergence of human and artificial intelligence. *Nature medicine*, 25(1):44–56, 2019.
- Umuroglu, Y., Fraser, N. J., Gambardella, G., Blott, M., Leong, P., Jahre, M., and Vissers, K. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74, 2017.
- Venieris, S. I. and Bouganis, C.-S. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 40–47. IEEE, 2016.
- Wang, E., Davis, J. J., Cheung, P. Y., and Constantinides, G. A. Lutnet: Learning fpga configurations for highly efficient neural network inference. *arXiv preprint arXiv:1910.12625*, 2019.
- Wu, J., Dong, M., Rigatto, C., Liu, Y., and Lin, F. Lab-on-chip technology for chronic disease diagnosis. *NPJ digital medicine*, 1(1):1–11, 2018.
- Xu, F., He, F., Xie, E., and Li, L. Fast obdd reordering using neural message passing on hypergraph. *arXiv preprint arXiv:1811.02178*, 2018.
- Zhao, R., Song, W., Zhang, W., Xing, T., Lin, J.-H., Srivastava, M., Gupta, R., and Zhang, Z. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA 17, pp. 1524, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450343541. doi: 10.1145/3020078.3021741. URL <https://doi.org/10.1145/3020078.3021741>.

Appendix

This section provides supplementary material with tables of experiment results of our pipeline.

6.1. Assumptions

The evaluation of our methods is based on the following assumptions and limitations. We only use fully-connected neural networks which we train with Adam optimizer (Kingma & Ba, 2014). For the GI bleeding data set, our architecture follows the one used in a broad evaluation of machine learning models on the same data set (Shung et al., 2020), which showed state-of-the-art results and was built out of only one hidden layer with 20 nodes. For training the real-valued neural network, we use a ReLU activation function on the input layer and on all hidden layers, while using a softmax activation function on the last layer in combination with a cross-entropy loss function. The ReLU activation function is also modeled within the logic of intermediate layers, but for the logic representation of the last layer, we use an identity mapping, followed by an argmax-function to get the final class prediction. This is done to prevent issues related to an exact approximation of the softmax function. Moreover we create only combinational circuits and leave the analysis and optimization of the timing aspects of the circuits as future work. Within the logic we make the assumption that the multiplier entities have the double bit width as the number of total bits used in the data-, the activation- and the weight-quantization scheme (see Section 4.3.1). Similarly, we assume all accumulator entities (which do the summations) to have three times of the bit width as the quantization scheme.

6.2. Further Results

The results of a broad experimental setup with multiple parameters can be found in the following.

Table 4. Results of **LogicNet** as intermediate step with the logic being quantized to **8 total bits and 6 fractional bits**. The column-wise “winner” is marked in bold.

DEPTH	WIDTH	LUT-SIZE	AIG NODES	AIG LEVELS	LOGIC ACCURACY (%)
2	50	4	80,756	187	54.17
2	100	4	71,162	186	54.00
2	200	4	56,738	184	54.17
2	50	6	107,190	188	54.17
2	100	6	116,563	188	54.17
2	200	6	176,307	204	54.17
2	50	8	224,670	202	53.50
2	100	8	217,562	204	54.33
2	200	8	135,649	189	54.00
3	50	4	239,243	205	53.50
3	100	4	221,350	201	53.83
3	200	4	215,211	206	54.17
3	50	6	146,347	192	56.50
3	100	6	249,995	203	54.00
3	200	6	147,908	196	54.33
3	50	8	149,378	197	54.17
3	100	8	147,145	195	55.83
3	200	8	149,399	196	54.17
4	50	4	146,130	192	54.33
4	100	4	146,592	191	54.17
4	200	4	145,878	191	50.33
4	50	6	254,011	209	54.67
4	100	6	151,359	197	54.17
4	200	6	255,886	205	58.83
4	50	8	255,403	209	54.33
4	100	8	256,910	212	56.00
4	200	8	157,274	201	57.33

Table 5. Results of **random forest** as intermediate step with the logic being quantized to **8 total bits and 6 fractional bits**. The column-wise “winner” is marked in bold.

ESTIMATORS	MAX. DEPTH	AIG NODES	AIG LEVELS	LOGIC ACCURACY (%)
2	5	198,378	227	54.17
2	10	541,789	251	48.50
2	15	472,091	261	53.50
3	5	341,500	272	52.50
3	10	737,429	263	52.00
3	15	630,701	275	54.17
4	5	266,826	252	55.00
4	10	912,589	275	49.83
4	15	793,930	287	48.83

Table 6. Results of the computing times for MiniSAT (Eén & Sörensson, 2003) to find satisfying arguments for multiple settings of bloated and reduced logic - ordered by the number of AIG nodes.

LOGIC TRANSLATION (m TOTAL BITS, i FRACTIONAL BITS)	SETTINGS	AIG NODES	AIG LEVELS	MINISAT TIME
NEURAL NETWORK (32, 16)	-	2,516,251	1,113	18,813.05 s (313.55 MIN)
RANDOM FOREST (8, 6)	ESTIMATORS: 4; MAX. DEPTH: 10	912,589	275	22,310.16 s (371.84 MIN)
LOGICNET (8, 6)	DEPTH: 4; WIDTH: 100; LUT-SIZE: 8	256,910	212	1,264.43 s (21.07 MIN)
NEURAL NETWORK (8, 4)	-	210,457	477	76.08 s (1.27 MIN)
RANDOM FOREST (8, 6)	ESTIMATORS: 2; MAX. DEPTH: 5	198,378	227	5,226.24 s (87.10 MIN)
LOGICNET (8, 6)	DEPTH: 2; WIDTH: 200; LUT-SIZE: 4	56,738	184	4,880.07 s (81.33 MIN)

Table 7. Results of **LogicNet** as intermediate step with the logic being quantized to **32 total bits and 16 fractional bits**. The column-wise “winner” is marked in bold.

DEPTH	WIDTH	LUT-SIZE	AIG NODES	AIG LEVELS	LOGIC ACCURACY (%)
2	50	4	207,153	422	54.17
2	100	4	208,996	431	54.17
2	200	4	858,284	493	56.33
2	50	6	1,519,223	511	47.33
2	100	6	1,437,803	497	54.50
2	200	6	756,667	453	54.17
2	50	8	984,625	454	51.50
2	100	8	948,956	454	51.17
2	200	8	1,031,077	450	53.67
3	50	4	1,744,909	515	46.83
3	100	4	1,590,823	514	54.17
3	200	4	716,599	453	53.50
3	50	6	238,3425	512	53.00
3	100	6	2,385,562	517	53.50
3	200	6	1,338,098	464	55.33
3	50	8	1,333,842	462	46.00
3	100	8	1,339,410	463	53.17
3	200	8	1,345,170	465	54.33
4	50	4	1,286,186	454	64.33
4	100	4	1,288,754	461	56.67
4	200	4	1,288,886	458	40.83
4	50	6	1,343,026	466	47.50
4	100	6	1,353,394	466	52.83
4	200	6	1,360,466	467	53.50
4	50	8	1,341,298	463	56.67
4	100	8	1,349,522	470	51.33
4	200	8	1,361,810	465	60.83

Table 8. Results of **random forest** as intermediate step with the logic being quantized to **8 total bits and 6 fractional bits**. The column-wise “winner” for a setting with and without AIG optimization is marked in bold.

ESTIMATORS	MAX. DEPTH	AIG NODES	AIG LEVELS	LOGIC ACCURACY (%)
2	5	1,660,911	506	54.67
2	10	2,688,533	530	52.50
2	15	2,389,484	541	51.67
3	5	1,865,767	519	55.67
3	10	4,164,037	592	48.50
3	15	2,934,422	553	51.67
4	5	3,153,180	600	53.33
4	10	3,458,848	565	50.00
4	15	3,469,420	564	52.67