Tiny Classifier Circuits: Evolving Accelerators for Tabular Data

Konstantinos Iordanou[1], Timothy Atkinson[2], Emre Ozer[3], Jedrzej Kufel[3], John Biggs[3], Gavin Brown[1], and Mikel Luján[1]

[1]Department of Computer Science, University of Manchester, UK
{firstname.lastname}@manchester.ac.uk
[2]NNAISENSE, Switzerland
{firstname.lastname}@nnaisense.com
[3]Pragmatic Semiconductor, Cambridge, UK
{eozer, jkufel, jbiggs}@pragmaticsemi.com

Abstract

A typical machine learning (ML) development cycle for edge computing is to maximise the performance during model training and then minimise the memory/area footprint of the trained model for deployment on edge devices targeting CPUs, GPUs, microcontrollers, or custom hardware accelerators.

This paper proposes a methodology for automatically generating predictor circuits for classification of tabular data with comparable prediction performance to conventional ML techniques while using substantially fewer hardware resources and power. The proposed methodology uses an evolutionary algorithm to search over the space of logic gates and automatically generates a classifier circuit with maximised training prediction accuracy. Classifier circuits are so tiny (i.e., consisting of no more than 300 logic gates) that they are called "Tiny Classifier" circuits, and can efficiently be implemented in ASIC or on an FPGA.

We empirically evaluate the automatic Tiny Classifier circuit generation methodology or "Auto Tiny Classifiers" on a wide range of tabular datasets, and compare it against conventional ML techniques such as Amazon's AutoGluon, Google's TabNet and a neural search over Multi-Layer Perceptrons. Despite Tiny Classifiers being constrained to a few hundred logic gates, we observe no statistically significant difference in prediction performance in comparison to the best-performing ML baseline. When synthesised as a Silicon chip, Tiny Classifiers use 8-18x less area and 4-8x less power. When implemented as an ultra-low cost chip on a flexible substrate (i.e., FlexIC), they occupy 10-75x less area and consume 13-75x less power compared to the most hardware-efficient ML baseline. On an FPGA, Tiny Classifiers consume 3-11x fewer resources.

1. Introduction

The relentless successes of Deep Neural Networks (DNNs), in achieving near (or better than) human accuracy for important application domains has created tremendous research and industrial momentum. Although originally much success was based on Convolutional Neural Networks and harnessing the availability of large labelled datasets of images, the successes have expanded to various other tasks and associated neural architectures (e.g., recurrent and transformers for Natural Language Processing). These large datasets are mainly images, audio or text. This kind of data can be characterised as homogeneous data.

Given the momentum gathered and the existence of common computational kernels across the different kinds of DNNs, we are witnessing a myriad of hardware accelerators for *inference* as well as *training* of DNNs. In both scenarios, the most common approach for these accelerators is to be programmable hardware with specialized datatypes and computations, rather than being a task-specific circuit. As DNNs have evolved, their computation has evolved from dense tensor operations towards increased sparsity.

To sum up, the current status quo separates the development of the specific DNN for a particular task from the process of developing the hardware accelerator for the training, or inference, of the specific DNN. In more general terms, the current best practice considers a Machine Learning (ML) technique which generates a model, where the training and the execution of the model versus the design and optimization of the hardware accelerator are isolated; at best a co-design happens. Nonetheless, both development activities intrinsically involve optimization processes. Thus, a reasonable question to postulate would be: *Could we develop a supervised learning technique that takes tabular data as input, and generates a circuit representation for classification behaving like an ML model?*

Our main contribution is to address this question by presenting a methodology to automatically generate classification circuits directly from tabular data. In contrast to homogeneous data (image, text), we focus on tabular data which, for example, can combine numerical and categorical data (heterogeneous). DNNs excel at capturing the spatial or semantic relationship in images or speech data. However, for tabular data, the correlation among the features is weaker, and the features have no intrinsic positional information. Hence, tabular data is an active research area for DNNs [10, 62, 63, 41].

Such heterogeneous data are ubiquitous [63], with many

associated real-world applications. The paper describing Google's TabNet [10] refers to tabular data as "the most common type of data in real world AI". Important use-cases for tabular data appear, for example, in healthcare [77], where various numerical and categorical descriptors of patients can be used to infer suggestions for medication [78, 16] and personalised treatments. Of particular relevance, tabular data often exists in resource-limited scenarios suited to low-power ML also known as tinyML [45, 55, 47, 44].

The fundamental reason behind the benefits of our methodology is two-fold. Firstly, our boolean function representation is otherwise known as a "decision tree", in ML, and thus inherits favourable properties of this representation. A series of recent studies has indicated that decision trees outperform Deep Learning on tabular data, notably Grinsztajn et al. [33] observe that tree-based models have a natural advantage in such data. Specifically, they explain:

> *"This superiority is explained by specific features of tabular data: irregular patterns in the target function, uninformative features, and non rotationally-invariant data where linear combinations of features misrepresent the information"* [33].

A second fundamental benefit of our approach is the method of learning the boolean function representation. Our proposed evolutionary scheme has the ability to bypass the local minima which may trap a traditional gradient-based tree boosting technique.

This paper proposes an alternative methodology to current ML and Deep Learning methods used in the prior art to make predictions from tabular data, and makes the following contributions:

1. We establish a connection for the first time between circuit synthesis and a supervised ML problem via Graph-Based Genetic Programming, a form of evolutionary computing. No previous graph-based genetic programming research, to our knowledge, has considered a hardware circuit representation to be an ML predictor.
2. We propose a methodology called "Auto Tiny Classifiers" to automatically generate hardware circuits from tabular data for ML classification using graph-based genetic programming. Tiny Classifier circuits are composed of a very small number of logic gates (i.e., a few hundred) and are capable of matching the prediction accuracy of the state-of-the-art ML classifiers that are less efficient in area and power when implemented in hardware.
3. We describe a toolflow that generates Tiny Classifiers as ASIC blocks. Then, we present the synthesis results of the Tiny Classifiers and ML baseline designs targeting the conventional Silicon technology. We also implement the Tiny Classifiers and ML baselines as FlexICs and fabricate them on flexible substrates (i.e., polyimide) using the flexible electronics fabrication technology. In addition, our toolflow generates Tiny Classifiers as Intellectual Property (IP) blocks so that they can be integrated as accelerators into a System-on-Chip (SoC). We demonstrate this in an Arm-based SoC with substantial FPGA resources in which the accelerators are synthesized.

Tiny Classifiers can be used in many scenarios; e.g., triggering circuits within an SoC [32]. A compelling scenario is to maintain an SoC in a low-power state while Tiny Classifiers are the always-on circuits. Once a situation of interest is uncovered by the classifier, then the rest (or subset) of the system would be awakened. Hardwired Tiny Classifiers can also be useful on their own for emerging Fast Moving Consumer Goods (FMCG) applications such as smart packaging enabled with flexible electronics (packages of dairy and meat products, labels of deodorant bottles, etc). Smart packages can be equipped with integrated circuits (ICs) fabricated on flexible substrates (e.g., plastic) using low-cost flexible electronics technology [53, 72, 17].

Flexible ICs are significantly less costly than Silicon-based ICs, paving the way to low-cost circuit customization [18]. Tiny Classifiers can be implemented as flexible ICs closely coupled with low-cost printed sensors in a smart package and can make in-situ real-time predictions. There are recent examples of proposing and demonstrating ML models as flexible ICs to make in-situ classifications [54, 56, 55] in emerging FMCG products. These are typical near-sensor computing system [79, 39] examples where a compute block is closely coupled with a sensor, and the sensor data are turned into knowledge in the form of inference immediately at the source by low-cost and energy-efficient hardware. The programmability of classifier circuits is not a requirement for smart packages because of short FMCG product lifetimes (e.g., days or weeks) where products along with their packages will be disposed/recycled after use.

The remainder of the paper is organized as follows. Section 2 provides a brief introduction to graph-based genetic programming. Section 3 discusses the adaptation of the evolutionary algorithm used in Auto Tiny Classifiers. Section 4 describes Auto Tiny Classifiers for generating Tiny Classifier circuits. Section 5 describes the tabular datasets used in the experimental evaluation and shows the performance, hardware design and implementation results. Finally, Section 6 presents the related work, and Section 7 concludes the paper.

2. Background

2.1. Graph-based Genetic Programming

The general graph-based genetic programming approach [51, 12, 21, 59] follows a traditional evolutionary methodology (see Figure 1). A set of possible solutions (the '*population*') are recombined ('*crossover*') and/or perturbed ('*mutation*'). The new, candidate, solutions (the '*children*') are then evaluated for their performance on the given task (giving a score, typically referred to as the '*fitness*'). The best-performing children are selected to form the new population in the next

iteration. Under the assumption that the problem has some sort of local continuity, such that children generated by performing crossover or mutation on high-quality solutions are more likely to be high-quality than randomly generated solutions, the algorithm tends towards higher-quality solutions over time. In doing so, it mimics natural Darwinian evolution, with the fitness acting as a selection pressure on the population, and mutation and crossover operators introducing variation.

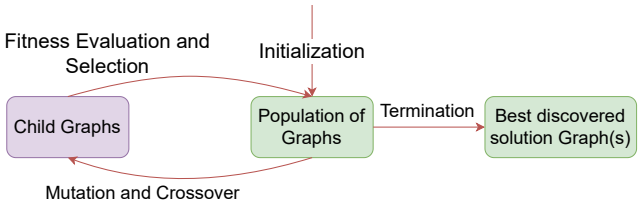

Figure 1: Overview of the Graph-based Genetic Programming methodology.

Graph-based genetic programming has been applied directly to both functional programs [46, 58] and stateful programs [20]. Graphs representing artificial neural networks have also been studied [9, 65]. The use of graph-based genetic programming for circuit synthesis has been considered in the literature [49, 48, 12, 64, 30], where the most prominent technique, Cartesian Genetic Programming (CGP), rooted in circuit synthesis has remained a relevant benchmark task [71, 36, 38]. Such existing studies typically consider the task of synthesis against a completely known truth table, even when working with approximate circuit synthesis [70, 52], where some error on that known truth table is acceptable in a tradeoff for greater efficiency.

In contrast, only a fraction of the truth table is known in our ML setting (tabular data classification), and the population consists of circuits, represented as graphs, which are evaluated for their ability to correctly classify the training data. The final performance is measured with respect to the ability of the generated circuit to generalise as measured with the unseen test data.

2.2. AutoML, NAS, NAIS versus Auto Tiny Classifiers

Figures 2, 3, 4 and 5 highlight the differences between current approaches of AutoML, Neural Architecture Search (NAS), Neural Architecture and Implementation Search (NAIS), and our Auto Tiny Classifier Circuits methodology for generating ML hardware as accelerators.

AutoML in Figure 2 and NAS in Figure 3 generate an ML model and a Neural Architecture model, respectively, with maximised prediction performance. However, the ML model must be translated into RTL, which, in turn, still needs to be verified. NAIS, in Figure 4, selects a specific Neural Network and a known Neural Network accelerator to iterate over the space, identifying the best parameters from the hardware pool to maximise the prediction accuracy.

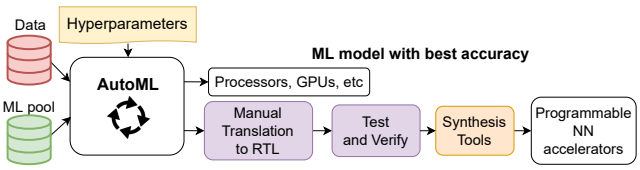

Figure 2: Automated Machine Learning (AutoML).

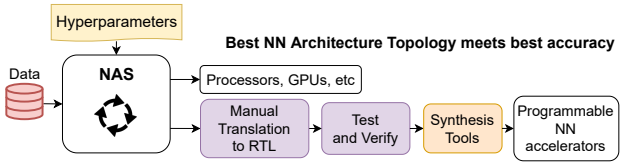

Figure 3: Neural Architecture Search (NAS).

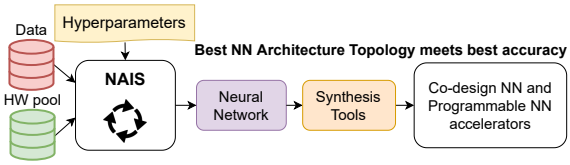

Figure 4: Neural Architecture and Implementation Search (NAIS).

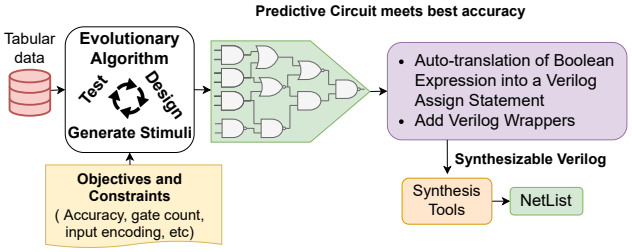

Figure 5: Auto Tiny Classifiers.

On the other hand, our proposed methodology searches the classifier circuit space automatically using an evolutionary algorithm, as shown in Figure 5. During circuit evolution, the generated circuit does not map to any predefined ML model or known hardware circuit. At the end of the search space cycle, the output is a sea of gates (a combinational circuit), which is auto-translated into RTL (i.e., typically as multiple Verilog assign statements for each classification output bit). These circuits are already verified during the fitness phase of the evolutionary algorithm. Our methodology is not a co-design approach, so there are no assumptions about any ML models or pre-determined hardware accelerator pools.

3. Automatically Evolving Classifier Circuits

Tabular data is addressed as a partial truth table of a circuit consisting of a sea of logic gates, where multiple heterogeneous features of the table are considered as the inputs to the circuit, whilst the classifier predictions are considered as its outputs. The fact that features in the tabular data are weakly correlated allows the conversion of the input-to-output prediction problem, into a simple representation of logic gates that can make predictions.

We adapt the Evolving Graphs by Graph Programming (EGGP) algorithm [12] as the evolutionary algorithm to generate the classification circuits. EGGP follows the consensus of using the simple $1+\lambda$ evolutionary technique [50], and particularly for circuit synthesis – a practice reinforced by recent empirical experiments [64, 30]. The algorithm consists of the following steps:

1. Generate a random initial parent solution S, and evaluate its fitness f_S.
2. While not terminated do:
   (a) Generate λ children $C_1 \dots C_\lambda$ by mutating S.
   (b) Evaluate the children's fitness values $f_1 \dots f_\lambda$.
   (c) If any child C_i has $f_i \geq f_S$, then replace the parent $S = C_i$, $f_S = f_i$. Where multiple children satisfy this condition, the child with the highest fitness is chosen, tie-breaks are determined at random.

The presence of the $\geq$ operator in the selection of a new parent, rather than just $>$, plays a pivotal role in the performance of the algorithm. In allowing the parent to be replaced by a child with equal fitness, the algorithm mimics the neutral drift of DNA as described in [42]. This allows the algorithm to undergo a 'random walk' in the space of equivalent solutions, to the best solution so far, exposing the algorithm to new neighbourhoods of possible children and thereby allowing it to escape local optima. This simple modification yields significant performance gains in practice [75, 76, 67] and may be augmented for further gains [13, 14, 24, 67], although we do not use these extensions in this work.

3.1. Solution Representation

In the algorithm, functional programs such as digital circuits are represented as graphs consisting of:

- A set of input nodes V_I, each node of which uniquely represents an input to the program.
- A set of function nodes V_F, each node of which represents a specific function applied to its inputs.
- A set of output nodes V_O, each node of which uniquely represents an output of the program.
- A set of edges E connecting function nodes and output nodes to their respective inputs.

While in general the edges of each node are ordered so that they appropriately handle commutative functions [12], in this case, all considered functions are symmetric.

A crucial property of the EGGP representation is that function nodes need not be 'active'. If there exists no path from a function node to an output node, then that node has no semantic meaning in the graph. This inactive material can be freely mutated to provide a direct mechanism for neutral drift.

3.2. Genetic Operators

When using the $1+\lambda$ evolutionary algorithm there are two main forms of genetic operator; initialisation and mutation.

Initialisation The initialisation is parameterized by the number of function nodes n, and the set of possible functions F. First, the I input nodes $i_1 \dots i_I$ are created. Then for each $i \in 1 \dots n$, a function node v_i is created and associated with a function chosen uniformly at random from F. v_i is then connected uniformly at random to existing nodes $i_1 \dots i_I, v_1 \dots v_{i-1}$ until its degree matches the number of expected inputs to f. Finally, the O output nodes $o_1 \dots o_O$ are created, and each is connected uniformly at random to a single node in $i_1 \dots i_I, v_1 \dots v_n$. The hyper-parameter n determines the overall size of the graphs throughout the duration of the evolutionary run.

Mutation Mutation on solutions is performed via point mutations drawn from binomial distributions. The mutation rate p parameterises the two binomial distributions $B(n,p)$ and $B(E,p)$ describing mutations of the functions nodes and edges, respectively. With $m_n \sim B(n,p)$ and $m_e \sim B(E,p)$ as the number of node and edge mutations to apply to the graph, the total $m_n + m_e$ mutations are applied in a randomly shuffled order, where;

- For node mutations, a random function node $v \in V_f$ is chosen, and its associated function f is replaced with $f' \in F, f' \neq f$ chosen uniformly at random. As the functions used here are symmetric and of the same arity, there is no need for input shuffling or connection modification procedures as described in [13].
- For edge mutations, a random edge $e \in E$ is chosen, where s is the source of e and t is the target of e. The edge is redirected such that its new target $v \in V_I \bigcup V_F$ is chosen uniformly at random where the following conditions hold:
  - There is no path $v \rightarrow s$ as this would introduce a cycle.
  - $v \neq t$ as this would not introduce any perturbation of the solution. In the special (very rare) case that the number of inputs $I = 1$ and there is only a single node $t = i_1$ satisfying the first condition, the mutation is abandoned.

3.3. Fitness

For all experiments performed here, the fitness of a circuit C is its balanced accuracy. In general, other fitness functions could be supported, including additional objectives such as the number of gates or power consumption, which could be handled through the use of multi-objective graph-based genetic programming [37] to search for the Pareto-optimal front of solutions and characterize the trade-off between the objectives. In experiments performed here, the evolutionary algorithm simply attempts to maximize the accuracy for a given dataset and has no prior knowledge of what the eventual prediction accuracy of the classifier circuit should be. Our methodology offers the option to split the data into training and validation sets (with a 50-50% split by default). During evolution, the fitness of circuits is evaluated on both the training and validation set separately. The fitness of the training set determines the selection of children to replace the parent, whereas the fitness

of the validation set ultimately determines the 'best-discovered solution'. Effectively, we are maximising performance on the training set, while using the validation set to attempt to identify the best-generalised solution. The performance reported later in this paper is the performance on the reserved (unseen) testing set, as described in Section 5.

3.4. Termination

In this setting, where the theoretical perfect accuracy of 100% may never be achieved, we require a termination condition. We use a simple model, whereby if the validation fitness (computed on the 50% validation set) has not improved by at least γ within κ generations, the algorithm terminates and returns the best-discovered solution with respect to the validation data. Additionally, the algorithm will automatically terminate if the number of generations exceeds the threshold G.

3.5. Hyperparameters

The hyperparameters of the algorithm are as follows:

- The number of children per generation, λ.
- The mutation rate, p.
- The function set from which solutions may be constructed, F.
- The termination threshold γ.
- The corresponding window of generations to achieve that threshold and terminate, κ.
- The maximum number of generations G.

In Section 5.3 we vary the function set F, number of function nodes n, termination generations κ and maximum number of generations G to choose hyper-parameters for evaluation in Section 5.4. The other hyper-parameters use the fixed values: $\lambda = 4$, $p = \frac{1}{n}$, $\gamma = 0.01$.

3.6. Classifier Circuits as Accelerators

The system can be thought as a set of classification circuit block(s) or a single classification circuit unit which lead to classification "guesses". The prediction could be a single bit (binary classification) or a set of bits in the case of multiclass classification problems which represent the encoding of the target class. Except for the actual classification circuit, the design uses buffers to hold the input and output data. The use of local buffers eliminates the data transfers within the system, keeping the required data close to the computation block(s).

Figure 6 presents classifier circuits as accelerators within a system. The inputs of the classifier circuit are single bits. The number of inputs for one classification circuit can be defined as the $number\_of\_features\_in\_one\_inference \times encoding\_bits\_per\_input$. Each feature of the inference is transformed into a group of bits based on the input encoding and the preferred number of bits per input. These parameters are user-defined. Most of the classification circuits use only a subset of input bits to perform a prediction. As a result, the above expression is the upper bound for an input size buffer. The actual size of the local buffer is determined after

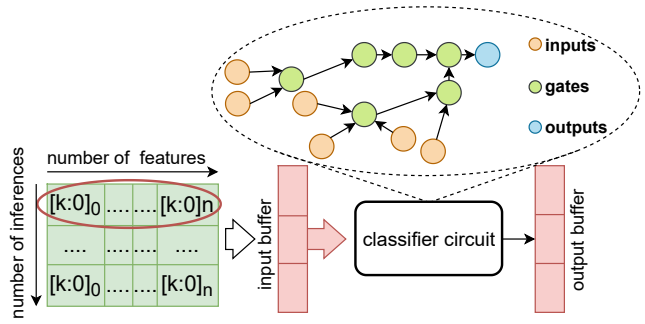

Figure 6: One instance of a classifier circuit.

the generation of the classification circuit and it holds only the necessary bits which will be consumed by the classification circuit for the prediction.

In the case of binary classification where the prediction is '0' or '1' ('yes' or 'no'), the output of the classifier is one bit. Basically, for each inference, we produce one classification and the result (single bit) is placed in the output buffer. However, for multiclass classification problems, the classification circuits have more than one output, which indicate the encoded predicted class. As a result, we instantiate $bits\_per\_output$ (user-defined parameter) local output buffers, which hold the encoded prediction for every inference. Of course, the size and the number of local input/output buffers increase the "cost" of the accelerator and this tradeoff should be explored based on the hardware specifications of the target embedded system.

Figure 6 presents the simplest accelerator which evolves classification circuits. It is the smallest possible accelerator design which includes a classification circuit. Identical classification circuits can be combined and process multiple inferences in parallel. In that case, the number of input local buffers is the number of parallel classification circuits within the accelerator. The processing of multiple inferences can be done in parallel, as long as there are available resources.

4. Auto Tiny Classifiers

Figure 7 shows the methodology of automatically generating Tiny Classifier circuits as hardware accelerators targeting ASIC and FPGA platforms.

4.1. From Tabular Data to Circuit Representations

The proposed methodology generates a visual representation of the classifier circuit directly from the training data and user-defined input parameters, as shown in Figure 7. The input parameters can be a subset or full set of the following; the total gate count of the classifier circuits, the type of the input encoding (binary, one-hot, gray), the number of required bits per input for the encoding and the quantization strategy (quantization/quantiles). The EGGP-based evolutionary algorithm crawls on the design space using the training data and converges on a simple graph of a sea of logic gates as the output circuit representation on which the test data is used to

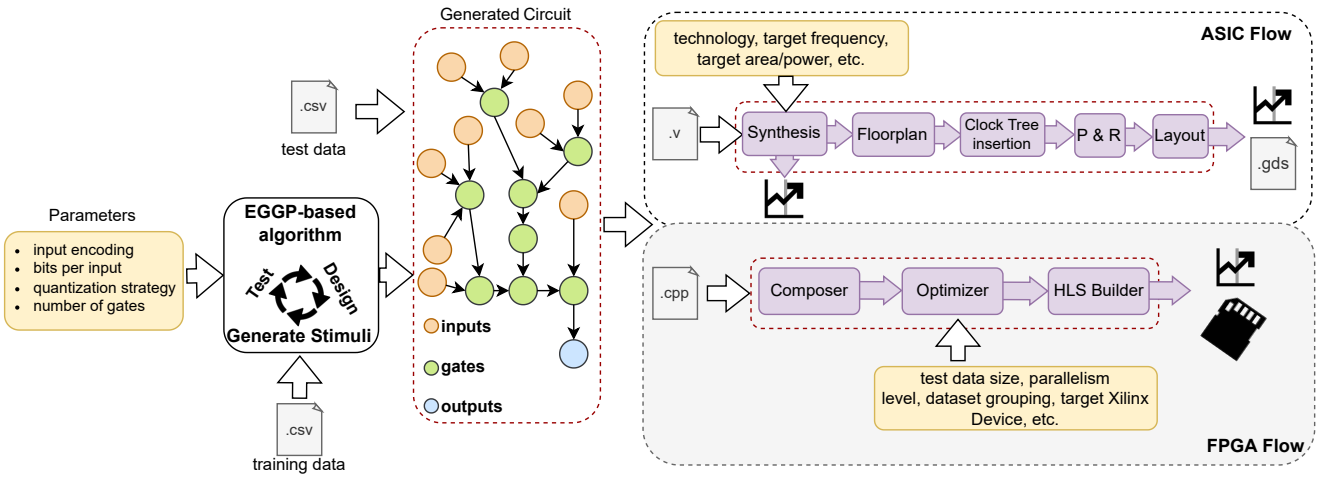

Figure 7: Methodology for generating Tiny Classifier circuits.

measure the final prediction accuracy. The sea of logic gates is automatically translated into RTL (e.g., Verilog).

4.2. ASIC and FPGA Target

Auto Tiny Classifiers generate Tiny Classifier circuits that can be implemented in ASIC, as shown in the dotted box inside the "ASIC flow" of Figure 7. The auto-generated Verilog representation of a Tiny Classifier is read by the synthesis tool that generates the netlist for a given technology standard cell library and constraints, and then produces the synthesized area, power and timing reports. The full chip implementation requires steps beyond synthesis such as floorplanning, clock tree insertion, place & route, and layout rules checking and generation. The output of the flow is the generation of the chip layouts in GDS format to complete the tape-out as well as the area, power and timing reports of the full implementation.

To target FPGAs, we use Xilinx SDSoC which is a software/hardware development environment from Xilinx for Zynq platforms [73] [74]. Xilinx SDSoC offers a complete software/hardware co-design environment where the designers develop the system software part along with a C/C++ implementation of the accelerated function, which will then be compiled by High-Level Synthesis (HLS). All layers between the host application and the hardware RTL (drivers, Operating Systems, etc.) are provided through an automated process. For our experiments, we use Xilinx Zynq Ultrascale+ MPSoC as a target platform which consists of a low-power processing system (PS) with a quad-core Arm Cortex-A53 coupled with a user-programmable logic part (PL). The generated Tiny Classifier circuit is the accelerator IP, which is then translated into C/C++ code for High-Level Synthesis to target an SoC platform partitioning the code between an FPGA target and a CPU [1]. After the transformation of a classifier circuit to a C/C++ function, the generated circuit is ready to be synthesized. This process is separated into three main phases: (a) Composer - gathers information about the generated classifier circuit and creates the necessary project files for the Xilinx SDSoC tools, (b) Optimizer - optimizes the Xilinx SDSoC generated project (`#pragma` directives for the HLS compiler, data transfer configuration between the PS and PL part of the target platform, etc) and (c) HLS Builder - produces a ready plug-and-play image with an integrated lightweight OS for the target platform including all the necessary libraries in a software/hardware co-design environment.

5. Evaluation

The experiments use a comprehensive collection of 33 tabular datasets, mainly from OpenML [69, 22, 29], UCI [25] and Kaggle [4]. For the datasets, we select the 17 used by Kadra *et al* which serve as a representative benchmark collection for tabular data [41], and in addition, focus on 16 mainly multiclass classification datasets from UCI, Kaggle and OpenML. For example, the dataset *higgs* contains sensor data from high-energy physics [15]. The dataset *clickpred* contains advertisements in a search engine, and whether or not they were clicked. From the selected collection, 14 tabular datasets were used by AutoGluon Tabular [26], the state-of-the-art AutoML tool for tabular data.

Table 1 provides the full list of datasets and their main characteristics. Each dataset is split into 80% training and 20% testing sets. The prediction accuracy results for both Tiny Classifier circuits and ML baseline models in the following subsections are based on test datasets.

5.1. Selection of Baseline ML Models

We use Google's TabNet DNN [10] with the recommended hyperparameters configuration, and AutoGluon (An AutoML system developed by Amazon) [26, 27] with explicit support for tabular data (Tabular Predictor) as well as other baseline

[1] It is also possible to use the generated Verilog module from our tool-flow to create a software/hardware co-design solution. However, all the layers (drivers, OS, etc.) must be developed manually without the automated process from Xilinx tools.

Dataset (Source)	Classes	Rows	Features
†vehicle (OpenML)	2	846	22
†cars (OpenML)	3	406	8
user model data (UCI)	4	403	5
†kc1 (OpenML)	2	145	95
†phoneme (OpenML)	2	5404	6
skin-seg (OpenML)	2	245057	4
ecoli-data (UCI)	4	336	8
iris (UCI)	3	150	7
†blood (OpenML)	2	748	4
†higgs (OpenML)	2	98050	29
wifi-localization (UCI)	4	2000	7
†nomao (OpenML)	2	34465	119
olinda-outlier (OpenML)	4	75	3
†australian (OpenML)	2	690	15
†segment (OpenML)	2	2310	20
led (UCI)	10	500	7
†numerai (OpenML)	2	96320	22
†miniboone (OpenML)	2	130064	51
wall-robot (Kaggle)	4	5456	3
†jasmine (OpenML)	2	2984	145
yeast (UCI)	10	1484	8
†christine (OpenML)	2	5418	1637
†sylvine (OpenML)	2	5124	21
seismic-bumps (UCI)	3	210	8
ccfraud (OpenML)	2	284807	31
clickpred (OpenML)	2	1496391	10
vowel (UCI)	2	528	21
nursery (UCI)	5	12958	9
spectf-data (Kaggle)	2	267	45
teaching assist (UCI)	3	151	7
wisconsin (UCI)	2	194	33
sonar (Kaggle)	2	208	61
ionosphere (UCI)	2	351	35

Note: †indicates that the dataset was appeared in the Auto-Gluon Tabular paper [26].

Table 1: The collection of the datasets.

ML models. Google's TabNet is one of the first successful DNNs addressing tabular data, using sequential attention to select features for decision-making layers. AutoGluon searches the design space over three state-of-the-art models (i.e, XG-Boost, TabNeuralNet and NNFastAITab) for Tabular Data among others. AutoGluon XGBoost is based on Gradient Boosting, whereas the other two models are based on DNNs. In our experiments, AutoGluon Tabular Predictor is configured with the above three models. *Kadra et al*. [41] observe that a Neural Architecture Search (NAS) over Multi-layer Perceptrons (MLPs) delivers state-of-the-art NN models for tabular data. Hence, we also use the NAS-based protocol described by *Kadra et al*. [41] to generate baseline MLP models.

5.2. Data Encoding and Quantization Strategy

Numerical inputs are automatically handled to encode the features of a dataset based on user preferences. The encoding consists of the encoding strategy and the number of bits per input. The encoding strategy determines the way that numerical features get translated into binary. Currently, four main encoding strategies are supported: (a) *quantization*, where each feature is divided into buckets of equal width, (b) *quantiles*, where each feature is divided into buckets of width roughly equal numbers to the number of samples, (c) one-hot and (d) gray. Additionally, the users can manually tune the *number of bits per input* to decide the granularity of the input encoding. From now onwards, experiments report only the best-achieved accuracy across the available encoding strategies with two and four bits per input.

In the comparative analysis with Tiny Classifiers, MLP models are transformed into 2-bit quantized versions. Since the hardware requirements of Tiny Classifiers are minimal, a comparison against the non-quantized MLPs does not provide a fair baseline when considering latency, area and power. Thus, we use a 2-bit quantized MLP as the resource-optimized high-performing baseline.

5.3. Tiny Classifier Design Space

A primary goal is to check whether we can generate accurate combinational logic for an ML classification problem. We explore different combinations of the hyperparameters (see Section 3.5) of the evolutionary algorithm to improve the accuracy of the generated circuits for all the datasets. Next, we explore the design space in four main directions: (a) the size of the generated circuit n (number of gates), (b) the function set F from which solutions (circuits) may be constructed, (c) the number of generations for the termination criterion function κ, and (d) the number of iterations to achieve a performance threshold and terminate G.

The heatmap of Figure 8a presents the achieved accuracy of the generated Tiny Classifier circuits as we progressively decrease the target NAND gate count from 300 to 50. At the same time, we explore the accuracy of the circuits with two different function sets. Overall, we observe a 14 percentage points reduction in GEOMEAN across all datasets from 300 gates to 50 gates.

The next step is to study how the number of generations for the termination criterion function impacts the accuracy of Tiny Classifiers when we limit the circuit size to a maximum of 300 gates. Figure 8b shows the achieved accuracy for various generation values of the termination criterion function. No significant change in prediction performance is observed.

Figure 8c presents the number of termination iterations versus achieved accuracy. We progressively increase the number of termination iterations as we set the target gate count and the number of generations for the termination function to 300. We observe a 2 percentage points improvement in GEOMEAN

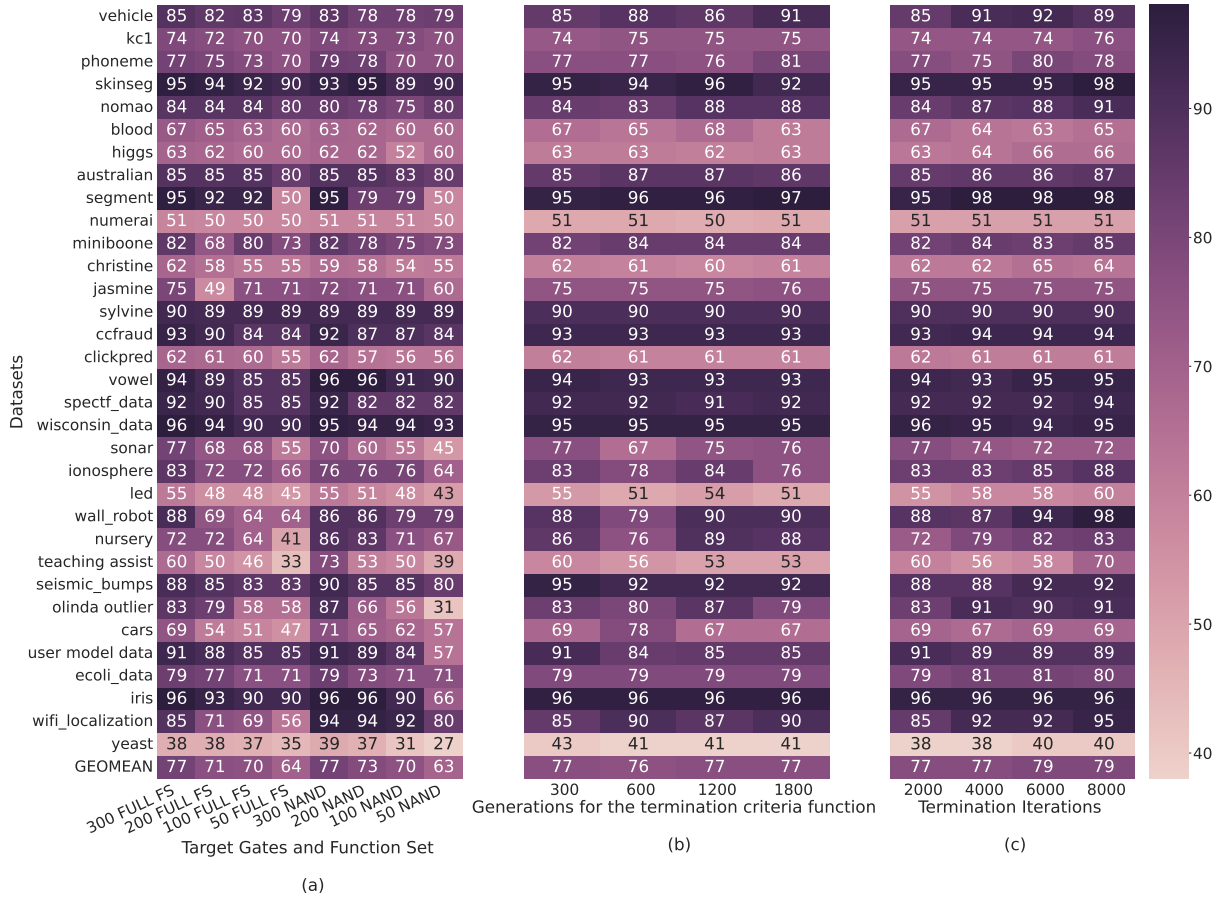

Figure 8: (a) Accuracy vs. number of gates. Generations for the termination function is 300 and termination iterations is 2000. *Full FS* indicates that the generated circuit will be constructed with logical gates within the function set $F = \{\texttt{and}, \texttt{or}, \texttt{nand}, \texttt{nor}\}$. For NAND function set the generated circuits constructed only with NAND gates. (b) Accuracy vs. generations for the termination function. The number of gates is 300 and the number of termination iterations is 2000. (c) Accuracy vs. the number of termination iterations. The number of gates and the generations for the termination function are both set to 300.

accuracy across all datasets when increasing the number of iterations.

5.4. Accuracy Comparison

Figure 9 compares the prediction accuracy of Google TabNet, AutoGluon and Tiny Classifiers. Based on the analysis in Section 5.3, the hyperparameters of Tiny Classifiers are set to 300 for both the number of gates and the termination function. In addition, the maximum number of iterations is set to 8000. Across all the datasets, the average prediction accuracy of AutoGluon XGBoost is 81%, which is the overall highest. The mean accuracy of Tiny Classifiers across all the datasets is 78%, which is the second highest.

We compare the prediction accuracy distribution of Tiny Classifiers against AutoGluon XGBoost to understand how robust Tiny Classifiers are with respect to XGBoost. To this end, we perform a 10-fold cross-validation study and show the accuracy distributions of Tiny Classifiers and XGBoost in Figure 10 using a violin plot.

The interquartile range of Tiny Classifiers is comparable to the interquartile ranges of ML baselines and in some cases, even slightly shorter. The shape of the distribution in Tiny Classifiers indicates that the accuracy data are highly concentrated around the median. This implies a low variance of the accuracy distribution and therefore makes Tiny Classifiers robust to variation.

The best-performing ML model, XGBoost, and Tiny Classifiers from Figure 9 are also compared to the best and smallest MLP configurations. We first explore the accuracy of a 9-layer MLP with 512 neurons following the protocol described in [41] (i.e., best MLP configuration) where the number of layers refers to the "hidden" layers of the neural network. The NAS takes this MLP as a starter and reduces the number of layers and neurons until reaching the smallest possible neural network size with minimal accuracy loss, which becomes a 3-layer MLP with 64 neurons.

Figure 11 shows the prediction accuracy of six models (i.e., XGBoost, Tiny Classifiers, non-quantized best MLP, 2-bit quantized best MLP, non-quantized smallest MLP and 2-bit quantized smallest MLP). Across all datasets, the non-

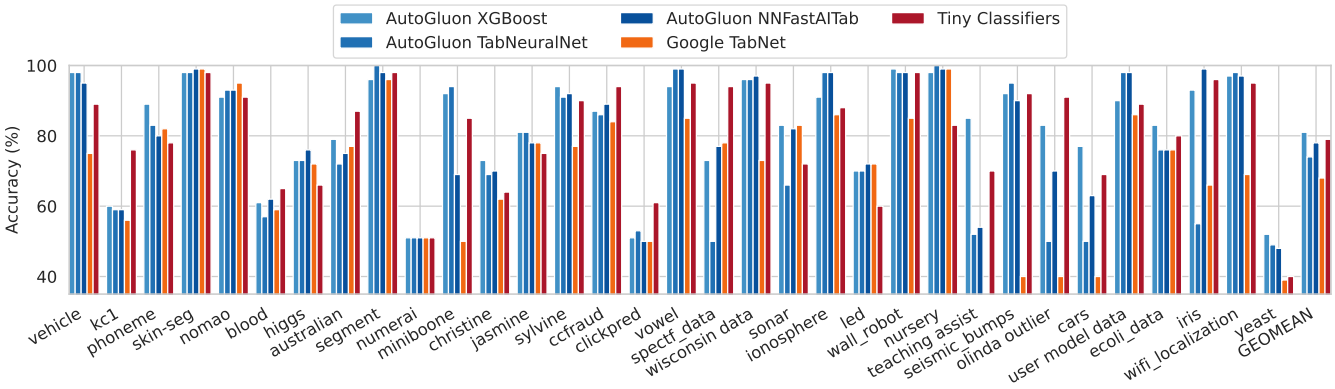

Figure 9: Prediction Accuracy of Tiny Classifiers, AutoGluon XGBoost, AutoGluon TabularNeuralNet, AutoGluon NNFastAITabular and Google TabNet. Note the datasets *vehicle* to *ionosphere* (left to right) are binary, and the remainder are multiclass classifications.

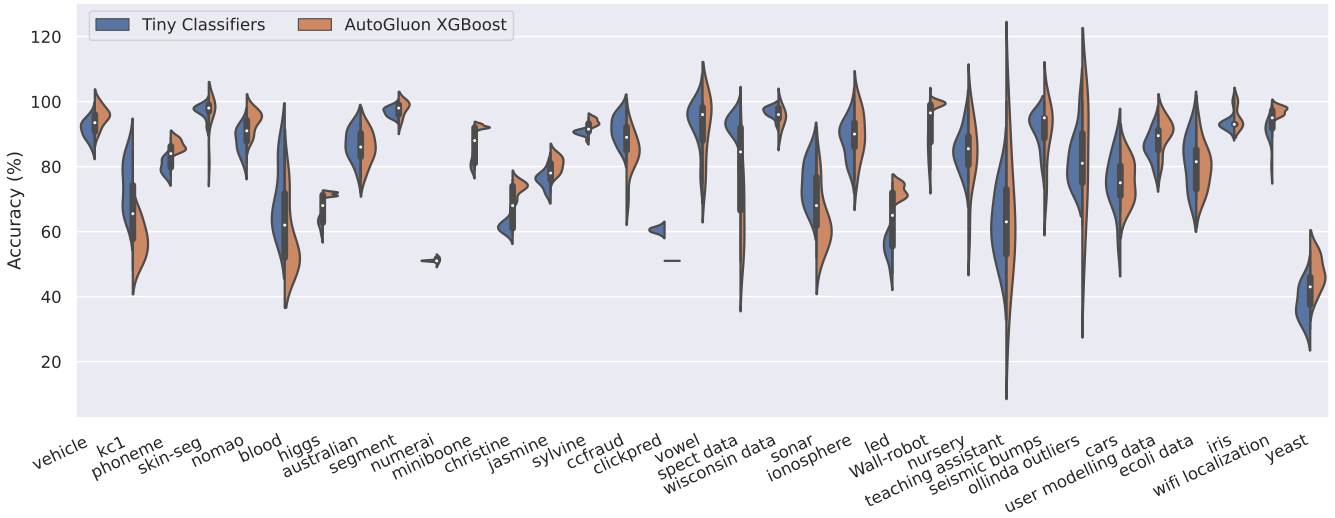

Figure 10: Violin plots showing the accuracy distributions of Tiny Classifiers and AutoGluon XGBoost.

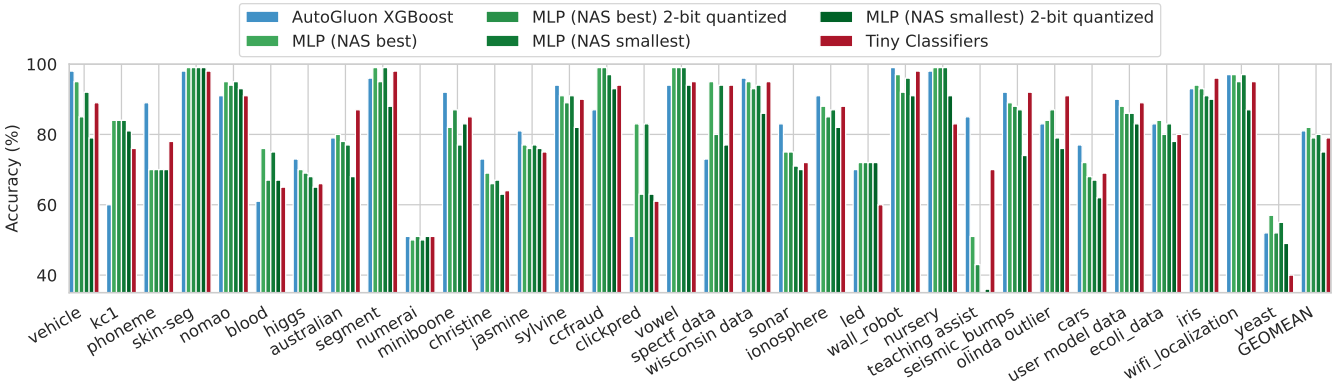

Figure 11: Prediction Accuracy of Tiny Classifiers, smallest MLP (non-quantized and 2-bit quantized versions), best MLP (non-quantized and 2-bit quantized versions) and AutoGluon XGBoost.

quantized best MLP model tops the performance by 83% overall prediction accuracy whilst its 2-bit quantized version has the same performance as Tiny Classifiers. In contrast, the non-quantized smallest MLP has an overall prediction accuracy of 80% whilst its 2-bit version stays at 75%. In summary, the performance of Tiny Classifiers is no worse than the 2-bit

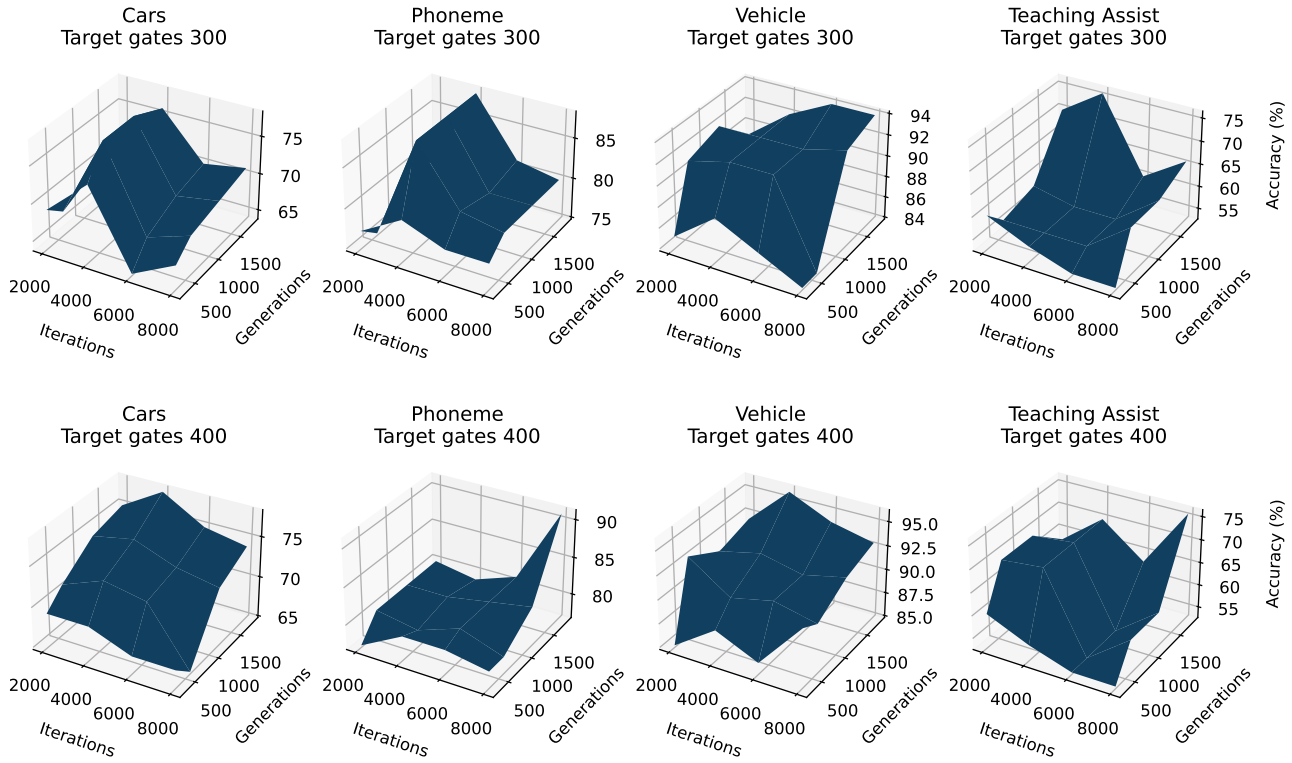

Figure 12: Space Exploration for four datasets (*vehicle, phoneme, Teaching Assist and Cars*) beyond the limit of 300 logical gates. The best achieved prediction accuracy of Tiny Classifier Circuits with 400 gates ranges from 76% - 96%.

quantized MLP models.

Figure 12 illustrates the benefit of increasing the circuit size limit from 300 gates to 400 gates of Tiny Classifiers for four datasets which present a poor classification accuracy compared to AutoGluon XGBoost. The prediction accuracy for these four datasets improves by up to 11 percentage points when moving the limit from 300 to 400 logical gates.

5.5. ASIC Flow Results

We design Tiny Classifiers in hardware across all datasets. For a comparison point, we also design the two ML baseline models in hardware. In addition to XGBoost (best performing ML baseline), the 2-bit quantized smallest MLP is also chosen as the second baseline ML model because it is the smallest MLP baseline (3 layers/64 neurons). As we needed to design the baseline ML models in hardware manually, we designed them only for two datasets (i.e., *blood* and *led*).

These two datasets are selected based on the number of classes and the complexity of implementing XGBoost in hardware. *blood* has one of the smallest numbers of classes (i.e., 2) and *led* has one of the largest numbers of classes (i.e., 10). The default number of estimators (Parallel Decision Trees) for XGBoost in Python [6] is 100 for a binary classification problem and $100 \times number\_of\_classes$ for multi-class classification. The number of estimators is strongly correlated with the achieved accuracy of the model. The main reason why *blood* is selected among other 2-class datasets is because XGBoost in *blood* has the smallest number of estimators with the smallest accuracy loss across all the 2-class datasets. A similar observation is made for *led* that it requires a smaller number of estimators compared to *yeast* (i.e., the other dataset that has also 10 classes) to achieve iso-performance.

One estimator (binary classification) for *blood* and 10 estimators (one estimator for each target class) for *led* are designed in hardware for XGBoost. For the development and the verification of the MLP and XGBoost designs, Bluespec System Verilog is used [2], and the designs are simulated with *Bluesim*.

5.5.1. Synthesis Results for Silicon Target: The Verilog representation of Tiny Classifiers and the two ML baselines are synthesized using Synopsis Design Compiler targeting the open 45nm PDK [31] Silicon technology. We present the synthesis power and area results for each Tiny Classifier circuit and baseline ML model as standalone hardware blocks, i.e., no interconnections to other components part of an overall ASIC design. The required data has been transferred to an input buffer, and the class predictions are stored in an output buffer inside the block. Both input and output buffers are included in the power and area calculations. The operational voltage and frequency are 1.1V and 1GHz, respectively.

Figures 14 and 15 show the power consumption and the area in NAND2-equivalent gate count. Tiny Classifier circuits consume 0.04 - 0.97 mW, and the gate count ranges from 11-426 NAND2-equivalent gates (combinational logic plus the I/O buffers). Note that two classifier circuits have just 3

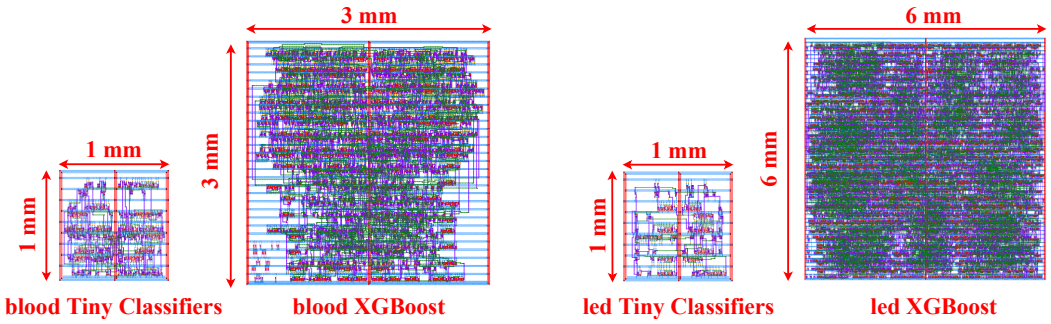

Figure 13: Flexible chip layouts of Tiny Classifiers and XGBoost implemented in PragmatIC's 0.8μm FlexIC TFT process for *blood* and *led* datasets.

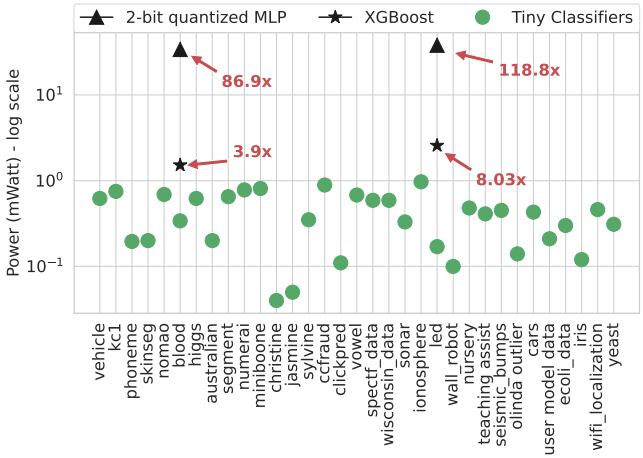

Figure 14: Power consumption of Tiny Classifier circuits across all datasets where MLP and AutoGluon XGBoost designs are shown for *blood* and *led* datasets.

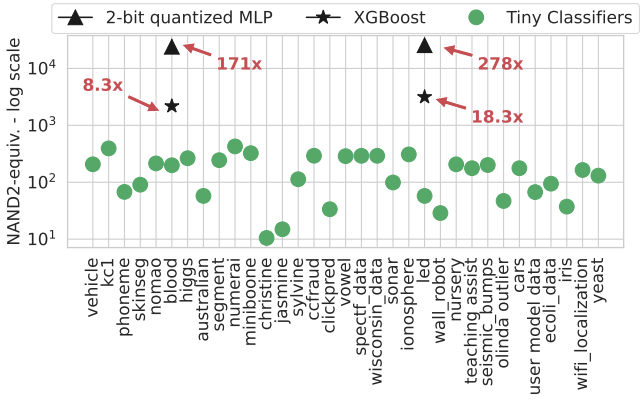

Figure 15: NAND2-equivalent gate count of Tiny Classifier circuits where MLP and AutoGluon XGBoost are shown for *blood* and *led* datasets.

gates excluding the I/O buffers. The power consumption of MLP is 34-38 mW (86-118 times greater than that of Tiny Classifiers), and the area is ∼171 and ∼278 times larger than Tiny Classifiers for *blood* and *led*. The power consumption of XGBoost is ∼3.9 and ∼8 times higher than Tiny Classifiers for *blood* and *led* whilst the area is 8 and 18 times larger than Tiny Classifiers, respectively.

5.5.2. Implementation Results for Flexible Chips: As discussed in *Introduction*, Tiny Classifiers are ideal for low-cost flexible chips for smart packages. We pick XGBoost as the ML baseline for comparison because it is more efficient in terms of occupied area and power than the MLP. Both Tiny Classifiers and XGBoost designs for *blood* and *led* are implemented with PragmatIC's 0.8μm FlexIC metal-oxide thin-film transistor (TFT) process in PragmatIC's FlexLogIC line [7]. The designs are put through the Cadence implementation flow to generate chip layouts[2].

Figure 13 shows the flexible chip layouts of the four designs. Table 2 summarizes the power, performance and area results. Tiny Classifier for *blood* is 10 times smaller and consumes about 13 times less power than XGBoost whilst it can run twice as fast as XGBoost. On the other hand, the comparative results for *led* are more prominent as Tiny Classifier is about 75 times smaller & lower power and three times faster than XGBoost. An important observation is that the area variation of Tiny Classifiers between a binary and a multi-class classification problem is negligible. Specifically, our methodology generates a smaller Tiny Classifier for *led* (105 NAND2-equiv. gates) compared to *blood* (150 NAND2-equiv. gates). In contrast, XGBoost implementation for *led* occupies 5 times more area than *blood* mainly due to the larger number of mapped estimators for multi-class classification.

	Tiny Classifiers		**XGBoost**	
	blood	*led*	*blood*	*led*
Cell Area (mm^2)	0.54	0.37	5.4	27.74
Power (mW)	0.32	0.25	4.12	18.6
Max. Freq. (kHz)	350	440	165	130
NAND2-equivalent	150	105	1520	7780

Table 2: Tiny Classifiers and XGBoost implementation results in PragmatIC's 0.8μm FlexIC TFT process at 3V supply voltage.

[2]Tiny Classifier and XGBoost designs for *blood* are sent for fabrication. They will be fabricated on a 30μm thick polyimide substrate and tested.

5.6. FPGA-based Comparison

We also prototype Tiny Classifiers, XGBoost and the 2-bit quantized smallest MLP for the two datasets on an FPGA platform to demonstrate the software-hardware co-design environment. Trained 2-bit quantized MLPs are synthesized on reconfigurable hardware using Xilinx FINN, the state-of-the-art tool which generates dataflow-style architectures of neural networks on FPGAs [19]. After the initial configuration of the MLPs, we use Brevitas [57] to transform the neural network to a quantized trained neural network. For the Brevitas training, we use 2-bit quantized ReLU activation functions and apply batch normalization between each layer and its activation. The configuration of Brevitas follows the recommendations of Xilinx FINN [68]. Then, Xilinx FINN is used to implement the trained neural network as a dataflow accelerator on FPGAs. We set the default configuration settings to build in *dataflow* mode.

Figure 16 presents the FPGA resource utilization on a Xilinx Zynq Ultrascale+ MPSoC. For *blood* dataset, we observe that Tiny Classifiers consume 2.43x less FPGA resources in terms of the number of look-up tables (LUTs) and flip-flops (FFs) when compared to XGBoost, and 10.7x less FPGA resources than the Smallest MLP. For *led* dataset, XGBoost and the Smallest MLP are 2.92x and 3.87x larger than Tiny Classifiers, respectively.

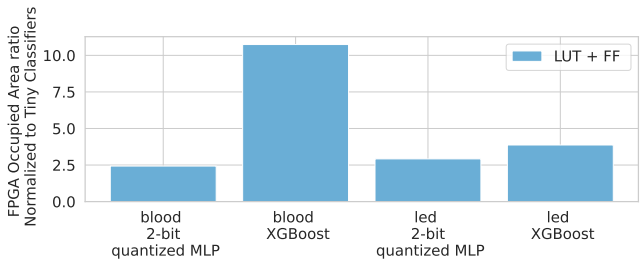

Figure 16: FPGA resource utilization (LUTs and FFs) of the 2-bit quantized Smallest MLP, XGBoost and Tiny Classifiers hardware implementation for *blood* and *led* datasets.

6. Related Work

Several methods have been proposed for supervised classification on tabular data. Two popular modern approaches are Gradient Boosted Decision Trees (GBDT), such as XGBoost [23] and CatBoost [61], and DNNs, such as TabNet by Google [10] and NODE [60]. Recent work on DNNs [60, 41] demonstrates that MLPs can be made competitive with state-of-the-art GBDT when the dimensions of the MLP architecture are suitably optimized. In addition, these optimized MLPs can also provide better accuracy than Google's TabNet [10]. Recall the full MLP exploration/optimization performed for our evaluation and associated accuracy results in Figure 11. This exploration ensures that the comparisons against the best MLP for accuracy, and resource utilization, against the smallest MLP, have optimized state-of-art baselines. Furthermore, using these baselines, our evaluation also facilitates the comparison of tiny classification circuits against their MLP counterparts on a well-established DNN accelerator.

Figures 2, 3 and 4 highlight the main features of AutoML, NAS-based and NAIS to generate ML-based hardware accelerators. AutoGluon is a prominent example of AutoML [26] [27] as well as H2O [3], AutoWeka [43], Auto-Sklearn [28], MLJAR [5] and Google Cloud AutoML Tables [1]. The current tools for AutoML do search the space for possible ML models (e.g. ensembles, DNNs, random forest) and these can be deployed for different inference tasks. Our experiments have used AutoGluon as a way to establish an optimized baseline for the accuracy of Neural Networks and XGBoost. However, the ML models generated by AutoML tools for tabular data do not generate RTL. That complex final step has to be done manually; see Figure 2. For NAS tools we find a dichotomy. On one hand, we find NAS tools which can handle tabular data, but only target standard processors, GPUs, and established programmable DNN accelerators (AutoGluon, Google Cloud AutoML). On the other hand, we can find those that cannot handle tabular data but can co-design a programmable Neural Network Accelerator (NAIS approach). These rely on known ML/NN model pools and known hardware architectures [8] [34] [35] [40]; most cases focusing on FPGAs. In the experiments, we have shown that a NAS exploration of MLPs for tabular data (as suggested by Kadra *et al.* [41]) produces accuracy results similar to or better than the NN produced by Amazon's AutoGluon and Google's TabNet while having the advantage of being smaller NNs.

Although our methodology aims to generate classifier circuits for tabular data, it is not in principle limited to tabular data. Work on recurrent graph-based genetic programming [66, 11] indicates the general applicability of the evolutionary approach to other forms of data, e.g. time-series data. Nonetheless, making progress with Graph-Based Genetic Programming in different data domains still remains a significant research challenge in its own right.

7. Conclusions

This paper proposes a methodology called "Auto Tiny Classifiers" to automatically generate classification circuits from tabular data. We have identified a connection between Graph-Based Genetic Programming with the classification problem in ML and proposed an evolutionary approach to generate Tiny Classifier circuits composed of a small number of logic gates (i.e., < 300 gates) and capable of matching the performance of the state-of-the-art ML techniques for tabular data.

We have evaluated the auto-generated Tiny Classifiers across 33 datasets and presented the synthesis results of Tiny Classifiers and ML baselines designed in ASIC in 45nm Silicon technology providing significant improvements in area/power. We have further implemented Tiny Classifiers and XGBoost (smallest ML baseline) as flexible chips using

0.8μm FlexIC TFT process technology. The full chip implementation results have shown that Tiny Classifiers could be clocked 2-3x faster and were 10-75x smaller and had lower power than XGBoost. We have also implemented Tiny Classifiers on an FPGA and demonstrated their area efficiency (3-11x fewer resources).

Thus, Tiny Classifiers can be integrated as tightly-coupled functional units or co-processors or become loosely-coupled hardware accelerators. Their smaller footprint and low power consumption make them attractive for near-sensor computing and emerging smart package applications.

8. Acknowledgements

Konstantinos Iordanou is funded by an Arm Ltd. & EPSRC iCASE PhD Scholarship. Mikel Luján is funded by an Arm/RAEng Research Chair award and a Royal Society Wolfson Fellowship. The research carried out by Timothy Atkinson happened while being an employee of the University of Manchester. The research is partially funded by EPSRC LAMBDA (EP/N035127/1) and EnnCore (EP/T026995/1), and UKRI NimbleAI (no. 10039070).

References

[1] AutoML Tables Documentation. https://cloud.google.com/automl-tables/docs.

[2] Bluespec System Verilog (BSV). http://wiki.bluespec.com/bluespec-systemverilog-and-compiler.

[3] H2o AutoML. https://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html.

[4] Kaggle. https://www.kaggle.com.

[5] MLJAR website. https://mljar.com/.

[6] XGBoost Documentation. https://xgboost.readthedocs.io/en/stable/.

[7] FlexLogIC. https://www.pragmaticsemi.com/create-more/devices, 2022.

[8] Mohamed S. Abdelfattah, Łukasz Dudziak, Thomas Chau, Royson Lee, Hyeji Kim, and Nicholas D. Lane. Best of both worlds: Automl codesign of a cnn and its hardware accelerator, 2020.

[9] Arbab Masood Ahmad and Gul Muhammad Khan. Bio-signal processing using cartesian genetic programming evolved artificial neural network (cgpann). In *10th International Conference on Frontiers of Information Technology*, pages 261–268. IEEE, 2012. doi: https://doi.org/10.1145/2463372.2463484.

[10] Sercan Ömer Arik and Tomas Pfister. Tabnet: Attentive interpretable tabular learning. *CoRR*, abs/1908.07442, 2019. arXiv: http://arxiv.org/abs/1908.07442.

[11] Timothy Atkinson. *Evolving graphs by graph programming*. PhD thesis, University of York, 2019.

[12] Timothy Atkinson, Detlef Plump, and Susan Stepney. Evolving graphs by graph programming. In *European Conference on Genetic Programming*, pages 35–51. Springer, 2018.

[13] Timothy Atkinson, Detlef Plump, and Susan Stepney. Evolving graphs with semantic neutral drift. *Natural Computing*, pages 1–17, 2019.

[14] Timothy Atkinson, Detlef Plump, and Susan Stepney. Horizontal gene transfer for recombining graphs. *Genetic Programming and Evolvable Machines*, 21(3):321–347, 2020.

[15] Sadowski P. Baldi P. and Whiteson D. Searching for Exotic Articles in High-Energy Physics with Deep Learning. *Nature Communications*, 5, 2014.

[16] Youjun Bao and Xiaohong Jiang. An intelligent medicine recommender system framework. In *2016 IEEE 11Th conference on industrial electronics and applications (ICIEA)*, pages 1383–1388. IEEE, 2016.

[17] John Biggs, James Myers, Jedrzej Kufel, Emre Ozer, Simon Craske, Antony Sou, Catherine Ramsdale, Ken Williamson, Richard Price, and Scott White. A natively flexible 32-bit Arm microprocessor. *Nature*, 595:532–536, 2021.

[18] Nathaniel Bleier, Calvin Lee, Francisco Rodriguez, Antony Sou, Scott White, and Rakesh Kumar. Flexicores: Low footprint, high yield, field reprogrammable flexible microprocessors. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, page 831–846, 2022.

[19] Michaela Blott, Thomas B. Preußer, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'Brien, and Yaman Umuroglu. FINN-R: an end-to-end deep-learning framework for fast exploration of quantized neural networks. *CoRR*, abs/1809.04570, 2018. arXiv: http://arxiv.org/abs/1809.04570.

[20] Markus Brameier and Wolfgang Banzhaf. Evolving teams of predictors with linear genetic programming. *Genetic Programming and Evolvable Machines*, 2(4):381–407, 2001.

[21] Markus F Brameier and Wolfgang Banzhaf. *Linear genetic programming*. Springer Science & Business Media, 2007.

[22] Giuseppe Casalicchio, Jakob Bossek, Michel Lang, Dominik Kirchhoff, Pascal Kerschke, Benjamin Hofner, Heidi Seibold, Joaquin Vanschoren, and Bernd Bischl. OpenML: An R package to Connect to the Machine Learning Platform Openml. *Computational Statistics*, 32(3):1–15, 2017.

[23] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[24] Richard Mark Downing. Neutrality and gradualism: encouraging exploration and exploitation simultaneously with binary decision diagrams. In *2006 IEEE International Conference on Evolutionary Computation*, pages 615–622. IEEE, 2006.

[25] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. url: http://archive.ics.uci.edu/ml.

[26] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505*, 2020.

[27] Rasool Fakoor, Jonas W Mueller, Nick Erickson, Pratik Chaudhari, and Alexander J Smola. Fast, accurate, and simple models for tabular data via augmented distillation. *Advances in Neural Information Processing Systems*, 33, 2020.

[28] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. *Auto-sklearn: Efficient and Robust Automated Machine Learning*, pages 113–134. Springer International Publishing, Cham, 2019.

[29] Matthias Feurer, Jan N van Rijn, Arlind Kadra, Neeratyoy Mallik Pieter Gijsbers, Sahithya Ravi, Andreas Mueller, Joaquin Vanschoren, and Frank Hutter. OpenML-Python: An Extensible Python API for OpenML. *arXiv*, 1911.02490, 2020.

[30] Léo Françoso Dal Piccol Sotto, Paul Kaufmann, Timothy Atkinson, Roman Kalkreuth, and Márcio Porto Basgalupp. Graph representations in genetic programming. *Genetic Programming and Evolvable Machines*, 22(4):607–636, 2021.

[31] FreePDK45. *Standard cell library 45nm*.

[32] Juan Sebastian P. Giraldo, Steven Lauwereins, Komail Badami, and Marian Verhelst. Vocell: A 65-nm speech-triggered wake-up soc for 10- μ w keyword spotting and speaker verification. *IEEE Journal of Solid-State Circuits*, 55(4):868–878, 2020.

[33] Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. Why do tree-based models still outperform deep learning on tabular data?, 2022.

[34] Cong Hao, Yao Chen, Xiaofan Zhang, Yuhong Li, Jinjun Xiong, Wen mei Hwu, and Deming Chen. Effective algorithm-accelerator co-design for ai solutions on edge devices, 2020.

[35] Cong Hao, Xiaofan Zhang, Yuhong Li, Sitao Huang, Jinjun Xiong, Kyle Rupnow, Wen mei Hwu, and Deming Chen. Fpga/dnn co-design: An efficient design methodology for iot intelligence on the edge, 2019.

[36] Simon L Harding, Julian F Miller, and Wolfgang Banzhaf. Self-modifying cartesian genetic programming. In *Cartesian Genetic Programming*, pages 101–124. Springer, 2011.

[37] James Hilder, James A Walker, and Andy Tyrrell. Use of a multi-objective fitness function to improve cartesian genetic programming circuits. In *2010 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 179–185. IEEE, 2010.

[38] David Hodan, Vojtech Mrazek, and Zdenek Vasicek. Semantically-oriented mutation operator in cartesian genetic programming for evolutionary circuit design. *Genetic Programming and Evolvable Machines*, 22(4):539–572, 2021.

[39] Ravi Iyer and Emre Ozer. Visual IoT: Architectural challenges and opportunities; Toward a self-learning and energy-neutral IoT. *IEEE Micro*, 36(6):45–49, 2016.

[40] Weiwen Jiang, Lei Yang, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Shouzhen Gu, Yiyu Shi, and Jingtong Hu. Hardware/software co-exploration of neural architectures. *CoRR*, abs/1907.04650, 2019.

[41] Arlind Kadra, Marius Lindauer, Frank Hutter, and Josif Grabocka. Regularization is all you need: Simple neural nets can excel on tabular data. *CoRR*, 2021. arXiv: https://arxiv.org/abs/2106.11189.

[42] Motoo Kimura. *The neutral theory of molecular evolution*. Cambridge University Press, 1983.

[43] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. *Auto-WEKA: Automatic Model Selection and Hyperparameter Optimization in WEKA*, pages 81–95. Springer International Publishing, Cham, 2019.

[44] Priyan Malarvizhi Kumar and Usha Devi Gandhi. A novel three-tier internet of things architecture with machine learning algorithm for early detection of heart diseases. *Computers & Electrical Engineering*, 65:222–235, 2018.

[45] Jongmin Lee, Michael Stanley, Andreas Spanias, and Cihan Tepedelenlioglu. Integrating machine learning in embedded sensor systems for internet-of-things applications. In *2016 IEEE international symposium on signal processing and information technology (ISSPIT)*, pages 290–294. IEEE, 2016.

[46] Juergen Leitner, Simon Harding, Alexander Forster, and Jurgen Schmidhuber. Mars terrain image classification using cartesian genetic programming. In *Proceedings of the 11th International Symposium on Artificial Intelligence, Robotics and Automation in Space, i-SAIRAS 2012*, pages 1–8. European Space Agency (ESA), 2012.

[47] Weixian Li, Thillainathan Logenthiran, Van-Tung Phan, and Wai Lok Woo. Implemented iot-based self-learning home management system (shms) for singapore. *IEEE Internet of Things Journal*, 5(3):2212–2219, 2018.

[48] Julian F Miller et al. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In *Proceedings of the genetic and evolutionary computation conference*, volume 2, pages 1135–1142, 1999.

[49] Julian F Miller, Peter Thomson, and Terence Fogarty. Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study. *Genetic algorithms and evolution strategies in engineering and computer science*, pages 105–131, 1997.

[50] Julian Francis Miller. Cartesian genetic programming: its status and future. *Genetic Programming and Evolvable Machines*, pages 1–40, 2019.

[51] Julian Francis Miller and Simon L Harding. Cartesian genetic programming. In *Proceedings of the 10th annual conference companion on Genetic and evolutionary computation*, pages 2701–2726, 2008.

[52] Vojtech Mrazek, Radek Hrbacek, Zdenek Vasicek, and Lukas Sekanina. Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 258–261. IEEE, 2017.

[53] Muhammad Husnain Mubarik, Dennis D. Weller, Nathaniel Bleier, Matthew Tomei, Jasmin Aghassi-Hagmann, Mehdi B. Tahoori, and Rakesh Kumar. Printed machine learning classifiers. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 73–87, 2020.

[54] Emre Ozer, Jedrzej Kufel, John Biggs, Gavin Brown, James Myers, Anjit Rana, Antony Sou, and Catherine Ramsdale. Bespoke machine learning processor development framework on flexible substrates. In *2019 IEEE International Conference on Flexible and Printable Sensors and Systems (FLEPS)*, pages 1–3, 2019.

[55] Emre Ozer, Jedrzej Kufel, John Biggs, James Myers, Charles Reynolds, Gavin Brown, Anjit Rana, Antony Sou, Catherine Ramsdale, and Scott White. Binary neural network as a flexible integrated circuit for odour classification. In *2020 IEEE International Conference on Flexible and Printable Sensors and Systems (FLEPS)*, pages 1–4, 2020.

[56] Emre Ozer, Jedrzej Kufel, James Myers, John Biggs, Gavin Brown, Anjit Rana, Antony Sou, Catherine Ramsdale, and Scott White. A hardwired machine learning processing engine fabricated with submicron metal-oxide thin-film transistors on a flexible substrate. *Nature Electronics*, 3(7):419–425, 2020.

[57] Alessandro Pappalardo. Xilinx/brevitas, 2021. doi: https://doi.org/10.5281/zenodo.3333552.

[58] Antonio Parziale, Rosa Senatore, Antonio Della Cioppa, and Angelo Marcelli. Cartesian genetic programming for diagnosis of parkinson disease through handwriting analysis: Performance vs. interpretability issues. *Artificial Intelligence in Medicine*, 111:101984, 2021.

[59] Riccardo Poli et al. Evolution of graph-like programs with parallel distributed genetic programming. In *ICGA*, pages 346–353. Citeseer, 1997.

[60] Sergei Popov, Stanislav Morozov, and Artem Babenko. Neural oblivious decision ensembles for deep learning on tabular data. *arXiv preprint arXiv:1909.06312*, 2019.

[61] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. Catboost: unbiased boosting with categorical features. *Advances in neural information processing systems*, 31, 2018.

[62] Stanislav Morozov Sergei Popov and Artem Babenko. Neural oblivious decision ensembles for deep learning on tabular data. *CoRR*, 2019. arXiv: http://arxiv.org/abs/1909.06312.

[63] Ravid Shwartz-Ziv and Amitai Armon. Tabular data: Deep learning is not all you need. *CoRR*, 2021. arXiv: https://arxiv.org/abs/2106.03253.

[64] Léo Françoso DP Sotto, Paul Kaufmann, Timothy Atkinson, Roman Kalkreuth, and Márcio Porto Basgalupp. A study on graph representations for genetic programming. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pages 931–939, 2020.

[65] Andrew James Turner and Julian Francis Miller. Cartesian genetic programming encoded artificial neural networks: a comparison using three benchmarks. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1005–1012, 2013.

[66] Andrew James Turner and Julian Francis Miller. Recurrent cartesian genetic programming. In *International Conference on Parallel Problem Solving from Nature*, pages 476–486. Springer, 2014.

[67] Andrew James Turner and Julian Francis Miller. Neutral genetic drift: an investigation using cartesian genetic programming. *Genetic Programming and Evolvable Machines*, 16(4):531–558, 2015.

[68] Xilinx FINN. Tutorial: Training and deploying a quantized mlp with xilinx finn, Dec 17, 2020. https://github.com/Xilinx/finn/tree/main/notebooks/end2end_example/cybersecurity.

[69] Joaquin Vanschoren, Jan N van Rijn, Bernd Bischl, and Luis Torgo. OpenML: Networked Science in Machine Learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

[70] Zdenek Vasicek and Lukas Sekanina. Evolutionary approach to approximate digital circuits design. *IEEE Transactions on Evolutionary Computation*, 19(3):432–444, 2014.

[71] James Alfred Walker and Julian Francis Miller. The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 12(4):397–417, 2008.

[72] Dennis D. Weller, Nathaniel Bleier, Michael Hefenbrock, Jasmin Aghassi-Hagmann, Michael Beigl, Rakesh Kumar, and Mehdi B. Tahoori. Printed stochastic computing neural networks. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 914–919, 2021.

[73] Xilinx. Xilinx SDSoC Development Environment Guide (UG1027). https://www.xilinx.com/support/documents/sw_manuals/xilinx2019_1/ug1027-sdsoc-user-guide.pdf.

[74] Xilinx. Xilinx SDSoC Programmers Guide (UG1278). https://www.xilinx.com/support/documents/sw_manuals/xilinx2018_2/ug1278-sdsoc-programmers-guide.pdf.

[75] Tina Yu and Julian Miller. Neutrality and the evolvability of boolean function landscape. In *European Conference on Genetic Programming*, pages 204–217. Springer, 2001.

[76] Tina Yu and Julian Miller. Finding needles in haystacks is not hard with neutrality. In *European Conference on Genetic Programming*, pages 13–25. Springer, 2002.

[77] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. Deep learning based recommender system: A survey and new perspectives. *ACM Comput. Surv.*, 52(1), feb 2019. doi: https://doi.org/10.1145/3285029.

[78] Yin Zhang, Daqiang Zhang, Mohammad Mehedi Hassan, Atif Alamri, and Limei Peng. Cadre: Cloud-assisted drug recommendation service for online pharmacies. *Mobile Networks and Applications*, 20(3):348–355, 2015.

[79] Feichi Zhou and Yang Chai. Near-sensor and in-sensor computing. *Nature Electronics*, 3(11):664–671, November 2020.