

Low-cost and efficient prediction hardware for tabular data using tiny classifier circuits

Received: 3 April 2023

Accepted: 22 March 2024

Published online: 25 April 2024

 Check for updates

Konstantinos Iordanou¹✉, Timothy Atkinson¹, Emre Ozer², Jędrzej Kufel², Grace Aligada², John Biggs², Gavin Brown¹ & Mikel Luján¹

A typical machine learning development cycle maximizes performance during model training and then minimizes the memory and area footprint of the trained model for deployment on processing cores, graphics processing units, microcontrollers or custom hardware accelerators. However, this becomes increasingly difficult as machine learning models grow larger and more complex. Here we report a methodology for automatically generating predictor circuits for the classification of tabular data. The approach offers comparable prediction performance to conventional machine learning techniques as substantially fewer hardware resources and power are used. We use an evolutionary algorithm to search over the space of logic gates and automatically generate a classifier circuit with maximized training prediction accuracy, which consists of no more than 300 logic gates. When simulated as a silicon chip, our tiny classifiers use 8–18 times less area and 4–8 times less power than the best-performing machine learning baseline. When implemented as a low-cost chip on a flexible substrate, they occupy 10–75 times less area, consume 13–75 times less power and have 6 times better yield than the most hardware-efficient ML baseline.

Deep neural networks (DNNs) can now offer near-human—or better than human—accuracy in a range of applications. Originally based on convolutional neural networks and harnessing the availability of large, labelled datasets of images, their application has expanded to many other tasks and associated neural architectures, such as recurrent and transformers for natural language processing. The large datasets used now are mainly images, audio or text, which can be characterized as homogeneous data. This progress, and the existence of common computational kernels across different kinds of DNN, has led to the development of a range 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 a task-specific circuit. As DNNs have evolved, their computation has evolved from dense tensor operations towards increased sparsity.

Currently, the training and execution of a machine learning (ML) model is typically separated from the design and optimization of the hardware accelerator, or—at best—some co-design happens. However,

both development activities involve optimization processes. Thus, a potential approach could be to develop a supervised learning technique that takes tabular data as the input and generates a circuit representation for classification that behaves like an ML model.

In this Article, we report a methodology—termed auto tiny classifiers—to automatically generate classification circuits directly from tabular data. In contrast to homogeneous data (image and text), we focus on tabular data that can, for example, 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 are an active research area for DNNs^{1–4}. Such heterogeneous data are ubiquitous², and have a range of practical applications^{5–7}. They also often exist in resource-limited scenarios suited to low-power ML, also known as tinyML^{8–11}.

Our approach offers an alternative methodology to current ML and deep learning methods for making predictions from tabular data

¹Department of Computer Science, University of Manchester, Manchester, UK. ²Pragmatic Semiconductor, Cambridge, UK.

✉ e-mail: konstantinos.iordanou@manchester.ac.uk

with two key benefits. First, our Boolean function representation is otherwise known in ML as a decision tree and thus inherits the favourable properties of this representation; recent studies have indicated that decision trees outperform deep learning on tabular data^{4,12}. Second, our evolutionary scheme can bypass the local minima that may trap a traditional gradient-based tree boosting technique.

Our tiny classifier circuits are composed of only a few hundred logic gates and can match the prediction accuracy of the state-of-the-art ML classifiers. We describe a toolflow that generates tiny classifiers as application-specific integrated circuit (ASIC) blocks. We then provide synthesis results of the tiny classifiers and ML baseline designs targeting conventional silicon technology. We also implement the tiny classifiers and ML baselines as flexible integrated circuits (FlexICs) and fabricate them on flexible substrates (polyimide).

The tiny classifiers could be used in a variety of applications. They could, for example, be used as triggering circuits within system on chips¹³, where the low-power state of a system on chip is maintained whereas the tiny classifiers are the always-on circuits. Hardwired tiny classifiers could also be used in fast-moving consumer goods applications such as smart packaging, where ML models on FlexICs^{14–17} can make in situ classifications^{9,18–20}. Smart packages can be equipped with integrated circuits (ICs) using low-cost flexible electronics technology^{14–16}. FlexICs are, in particular, less costly than silicon-based ICs, offering low-cost circuit customization¹⁷. Likewise, they are of potential value in low-cost near-sensor computing systems^{21,22} where a compute block is closely coupled with a sensor, and the sensor data are turned into knowledge using inference at the source. The programmability of classifier circuits is also not a requirement for smart packages due to short fast-moving consumer goods product lifetimes (days or weeks, for example), where the products, along with their packages, will be disposed/recycled after use.

Graph-based genetic programming

The general graph-based genetic programming approach^{23–26} follows a traditional evolutionary methodology (Fig. 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 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 of a higher quality than randomly generated solutions, the algorithm tends towards higher-quality solutions over time. Consequently, it mimics natural Darwinian evolution, with the fitness acting as selection pressure on the population, and mutation and crossover operators introducing variation.

Graph-based genetic programming has been directly applied to both functional^{27,28} and stateful programs²⁹. The use of graph-based genetic programming for circuit synthesis has been considered in the literature^{24,30–33}, where the most prominent technique, namely, Cartesian genetic programming, rooted in circuit synthesis has remained a relevant benchmark task^{34–36}. Such studies typically consider the task of synthesis against a completely known truth table, even when working with approximate circuit synthesis^{37,38}. In contrast, only a fraction of the truth table is known in our ML setting, and the population consists of circuits represented as graphs.

Figure 2 highlights the differences between the 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 (Fig. 2a) and NAS (Fig. 2b) generate an ML model and neural architecture model, respectively, with maximized prediction performance. However, the ML model must be translated into register-transfer level (RTL), which, in turn, still needs to be verified.

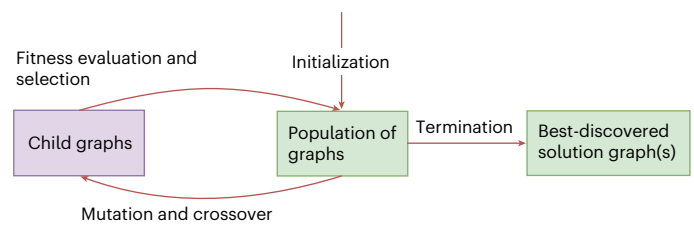


Fig. 1 | Overview of the graph-based genetic programming methodology. A set of possible solutions (population) are recombined (crossover) and/or perturbed (mutation). The new, candidate solutions (children) are then evaluated for their performance on the given task (fitness).

NAIS (Fig. 2c) 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 maximize the prediction accuracy.

On the other hand, our proposed methodology automatically searches the classifier circuit space using an evolutionary algorithm (Fig. 2d). During circuit evolution, the generated circuit does not map to any predefined ML model or a known hardware circuit. At the end of the search-space cycle, the output is a sea of gates (a combinational circuit), which is autotranslated into RTL. These circuits are already verified during the fitness phase of the evolutionary algorithm. Our methodology is not a co-design approach; therefore, there are no assumptions about any ML models or pre-determined hardware accelerator pools.

Several methods have been proposed for supervised classification on tabular data. Two popular modern approaches are gradient-boosted decision trees, such as XGBoost³⁹ and CatBoost⁴⁰, and deep learning architectures, such as TabNet by Google¹. Recent work on DNNs⁴ demonstrates that multilayer perceptrons (MLPs) can be made competitive with state-of-the-art gradient-boosted decision trees when the dimensions of the MLP architecture are suitably optimized. In addition, these optimized MLPs can also provide better accuracy than TabNet¹.

Automatically evolving classifier circuits

The fact that the features of tabular data are weakly correlated allows the conversion of the input-to-output prediction problem, into a simple representation of logic gates making predictions. We adapt the evolving graphs by graph programming (EGGP) algorithm²⁴ as the evolutionary algorithm to generate the classification circuits. EGGP follows the consensus of using the simple $1 + \lambda$ evolutionary technique⁴¹, particularly for circuit synthesis^{32,33}. The algorithm mimics the neutral drift of DNA⁴² and 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 $S = C_i$, $f_S = f_i$.
The point at which multiple children satisfy this condition, the child with the highest fitness is chosen; tie-breaks are determined at random.

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 a program input.
- 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 a program output.
- A set of edges E connecting function and output nodes to their respective inputs.

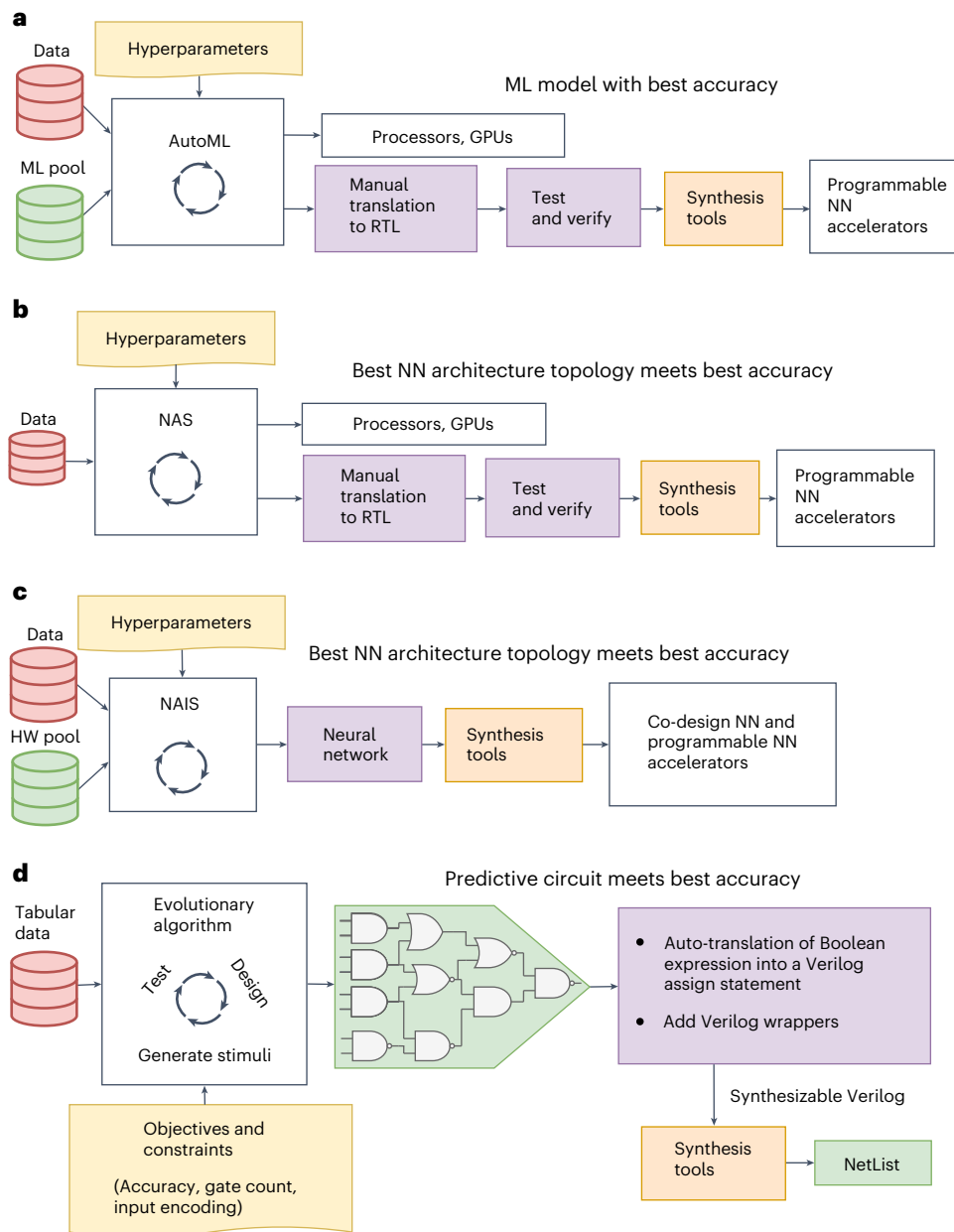


Fig. 2 | Differences between current approaches of AutoML, NAS, NAIS and our auto tiny classifier circuits. a, b. AutoML (a) and NAS (b) generate an ML model and a neural architecture model, respectively, with maximized prediction performance. However, the ML model must be translated into RTL and verified. **c.** NAIS selects a specific neural network (NN) and a known neural network

accelerator to iterate over the space, identifying the best parameters from the hardware (HW) pool to maximize the prediction accuracy. **d.** Our proposed methodology automatically searches the classifier circuit space using an evolutionary algorithm. During circuit evolution, the generated circuit does not map to any predefined ML model or known hardware circuit.

Although, in general, the edges of each node are ordered so that they appropriately handle commutative functions²⁴, in this case, all the considered functions are symmetric. A crucial property of the EGGP representation is that the 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.

When using the $1 + \lambda$ evolutionary algorithm, there are two main forms of genetic operator: initialization and mutation. The initialization 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_n are created. Then, for each $i \in 1 \dots n$, a function node v_i is created and associated with a function uniformly chosen at random from F . Also, v_i is then uniformly connected at random to the existing nodes $i_1, \dots, i_n, v_1, \dots, v_{i-1}$ 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 uniformly connected at random to a single node in $i_1, \dots, i_n, v_1, \dots, v_n$. The hyperparameter n determines the overall size of the graphs throughout the duration of the evolutionary run.

Mutation on solutions is performed via point mutations drawn from binomial distributions. The mutation rate p parameterizes the two binomial distributions $B(n, p)$ and $B(E, p)$ describing the mutations of function nodes and edges, respectively. With $m_n \approx B(n, p)$ and $m_e \approx B(E, p)$ as the number of node and edge mutations to apply to the graph, respectively, 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 $f' \in F, f' \neq f$ uniformly

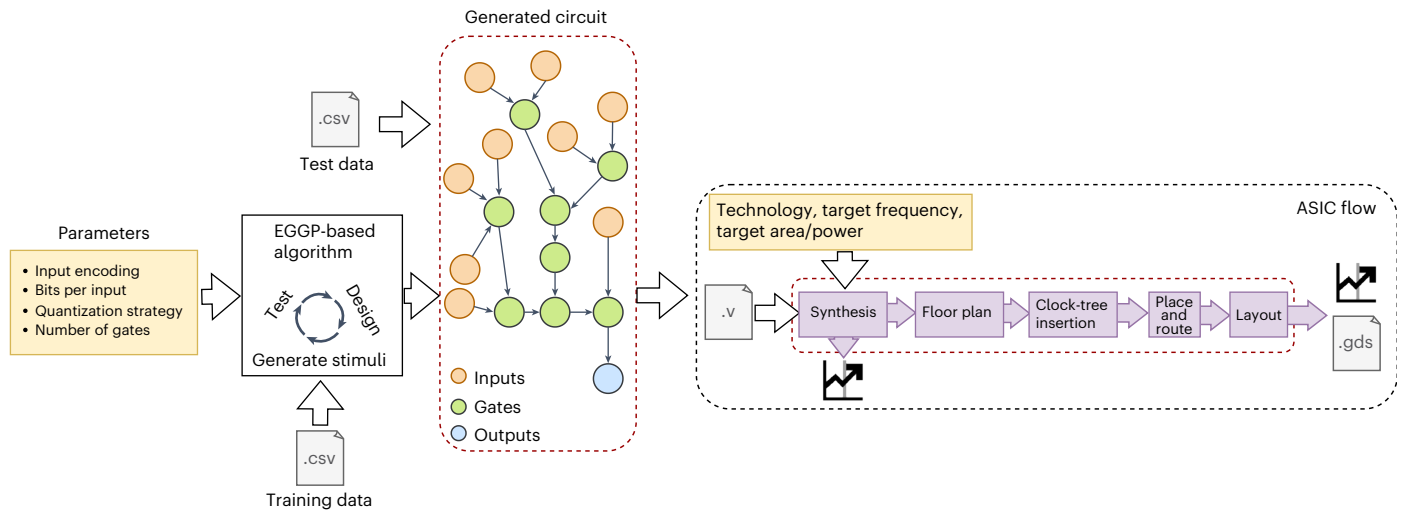


Fig. 3 | Methodology for generating tiny classifier circuits. Auto tiny classifiers directly generate a visual representation of the classifier circuit from the training data and user-defined input parameters. The EGGP-based evolutionary algorithm crawls on the design space using the training data and converges on a

simple graph of a sea of logic gates as the output circuit representation, which is automatically translated into RTL. The output of the flow is the generated chip layouts in GDS format to complete the tape-out as well as the area, power and timing reports of synthesis and full implementation.

chosen at random. As the functions used here are symmetric and of the same arity, there is no need for input shuffling or connection modification procedures⁴³;

- 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 $v \in V_1 \cup V_r$ is uniformly chosen at random where the following conditions hold:
 - There is no path $v \rightarrow s$ to avoid cycles.
 - $v \neq t$ 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.

For all the experiments performed here, the fitness of a circuit C is its balanced accuracy. In general, other fitness functions could be supported, including objectives such as the number of gates or power consumption, which could be handled through the use of multiobjective graph-based genetic programming to search for the Pareto-optimal front of solutions and characterize the trade-off between objectives. The evolutionary algorithm simply attempts to maximize the accuracy for a given dataset with no prior knowledge of the eventual prediction accuracy of the classifier circuit.

During evolution, the fitness of the circuits is separately evaluated on both training and validation set. 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 maximizing the performance on the training set, and the validation set is used to attempt to identify the best-generalized solution. The performance reported later in this Article is the performance on the reserved (unseen) testing set.

In the termination setting, 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 .

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 this Article, we vary the function set F , number of function nodes n , termination generations κ and maximum number of generations G to choose the hyperparameters for evaluation. The other hyperparameters use fixed values: $\lambda = 4$, $p = \frac{1}{n}$, $\gamma = 0.01$.

Auto tiny classifiers

Figure 3 shows the methodology of automatically generating tiny classifier circuits as hardware accelerators. Auto tiny classifiers directly generate a visual representation of the classifier circuit from the training data and user-defined input parameters. Input parameters can be a subset or full set of the following: the total gate count of the classifier circuits, the type of input encoding (binary, one-hot, gray), the number of required bits per input for the encoding and 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, which is automatically translated into RTL.

The autogenerated Verilog representation of a tiny classifier is read by the synthesis tool generating 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 additional steps such as floor planning, clock-tree insertion, place and route, and checking and generating the layout rules. The output of the flow is the generated chip layout in the GDS format to complete the tape-out as well as the area, power and timing reports of the full implementation.

The generated hardware can be thought of as a set of classification circuit block(s) or a single classification circuit unit that leads to classification 'guesses'. The prediction could be a single bit (binary classification) or a set of bits in the case of multiclass classification problems representing the target class. Except for the actual classification circuit, the design uses buffers to hold the input and output data. Local buffers eliminate the data transfers within the system, keeping the required data close to the computation block(s).

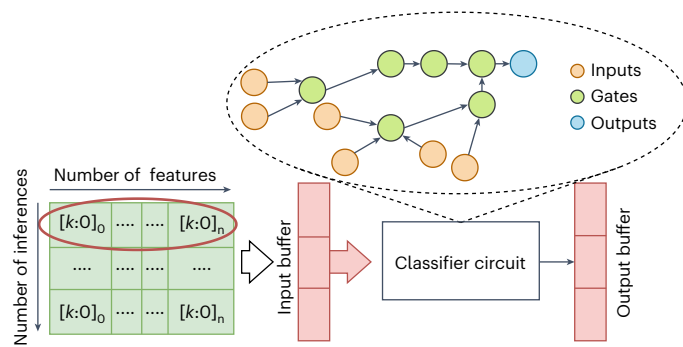


Fig. 4 | A classifier circuit as a hardware accelerator within a system. The system can be thought of as a single classification circuit unit, which leads to classification ‘guesses’. The prediction could be a single bit (binary classification) or a set of bits in the case of multiclass classification 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.

Figure 4 presents the classifier circuit as an accelerator within a system. The inputs to 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. The actual size of the local input buffer is determined after the classification circuit generation and it holds the input 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 classifier output 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, indicating the encoded predicted class. As a result, we instantiate bits_per_output (a user-defined parameter) local output buffers, which hold the encoded prediction for every inference.

Evaluation

The experiments use a comprehensive collection of 33 tabular datasets, mainly from OpenML⁴⁴, UCI⁴⁵ and Kaggle⁴⁶. Extended Data Table 1 provides the full list of datasets and their main characteristics. Each dataset is split into 80% training and 20% testing sets.

We use Google’s TabNet architecture¹ with the recommended hyperparameter configuration, and AutoGluon (an AutoML system developed by Amazon) with explicit support for tabular data (tabular predictor)⁴⁷ as well as other baseline ML models.

Google’s TabNet is one of the first successful deep learning architectures 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 (namely, XGBoost, 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. It has been observed⁴ that an NAS over MLPs delivers state-of-the-art NN models for tabular data. Hence, we also use this NAS-based protocol.

Numerical inputs are automatically handled by ‘auto tiny classifiers’ to encode the dataset features 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: (1) quantization, where each feature is divided into buckets of equal width; (2) quantiles, where each feature is divided into buckets of width roughly equal 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 two-bit quantized versions. Since the hardware requirements of tiny classifiers are minimal, we use a two-bit quantized MLP as the resource-optimized high-performing baseline.

Our primary goal is to check whether we can generate accurate combinational logic for an ML classification problem. We explore different hyperparameter combinations. The heat map shown in Extended Data Fig. 1a presents the achieved accuracy of the generated tiny classifier circuits as we progressively decrease the target NAND gate count from 300 to 50. Simultaneously, we explore the accuracy of the circuits with two different function sets. 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. Extended Data Fig. 1b shows the achieved accuracy for various generation values of the termination criterion function. Extended Data Fig. 1c 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.

Figure 5a compares the prediction accuracy of Google TabNet, AutoGluon and tiny classifiers. The hyperparameters of tiny classifiers are set to 300 for the number of gates as well as the termination function. In addition, the maximum number of iterations is set to 8,000 (Extended Data Fig. 1). Across all the datasets, the average prediction accuracy of AutoGluon XGBoost is 81%, which is the highest overall. 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. Thus, we perform a tenfold cross-validation study and show the accuracy distributions of tiny classifiers and XGBoost (Fig. 5b). The distribution shape in tiny classifiers indicates a low variance of the accuracy distribution and therefore makes tiny classifiers robust to variation.

The best-performing ML model, XGBoost and tiny classifiers (Fig. 5a) are also compared with the best and smallest MLP configurations. We first explore the accuracy of a nine-layer MLP with 512 neurons⁴. 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, becoming a three-layer MLP with 64 neurons.

Figure 5c shows the prediction accuracy across the six described models. Across all the datasets, the non-quantized best MLP model tops the performance by 83% overall prediction accuracy, whereas its two-bit quantized version has the same performance as tiny classifiers. In contrast, the non-quantized smallest MLP has an overall prediction accuracy of 80%, whereas its two-bit version stays at 75%. In summary, the performance of tiny classifiers is no worse than the two-bit quantized MLP.

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

These two datasets are selected on the basis of the number of classes and the complexity of implementing XGBoost in hardware. Blood has one of the smallest numbers of classes (that is, two) with the smallest accuracy loss across all the two-class datasets and led has one of the largest numbers of classes (that is, ten). One estimator (binary classification) for blood and ten estimators (one estimator for each target class) for led are designed in hardware for XGBoost.

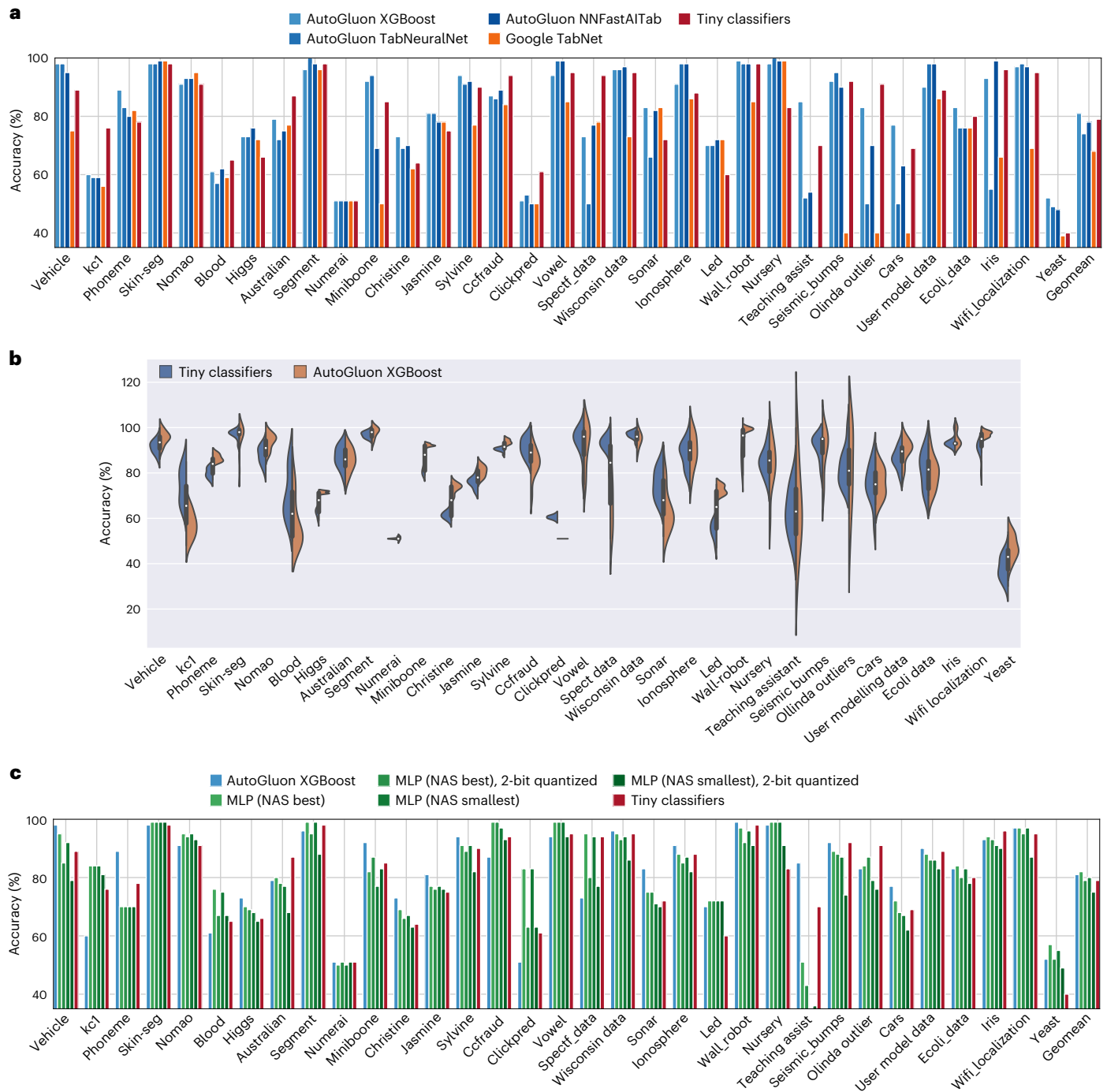


Fig. 5 | Prediction accuracy analysis of tiny classifiers. **a, b**, Prediction accuracy (**a**) of tiny classifiers compared with AutoGluon XGBoost, AutoGluon TabularNeuralNet, AutoGluon NNFastAITabular and Google TabNet and (**b**) robustness of tiny classifiers and AutoGluon XGBoost. **c**, Prediction accuracy

of tiny classifiers compared with the smallest MLP (non-quantized and two-bit quantized versions), best MLP (non-quantized and two-bit quantized versions) and AutoGluon XGBoost.

For the development and verification of the MLP and XGBoost designs, Bluespec System Verilog is used.

The Verilog representations of tiny classifiers and the two ML baselines are synthesized using Synopsis Design Compiler targeting the open 45 nm PDK⁴⁸ silicon technology. We present the synthesis power and area results for each tiny classifier circuit and baseline ML model as standalone hardware blocks, that is, no interconnections to other components of an overall ASIC design. Both input and output buffers are included in the power and area calculations. The operational voltage and frequency are 1.1 V and 1 GHz, respectively.

Figure 6a,b shows the power consumption and area in NAND2-equivalent gate count. Tiny classifier circuits consume 0.04–0.97 mW, and the gate count ranges from 11 to 426 NAND2-equivalent gates. 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 ~8.0 times higher than tiny classifiers for blood and led, whereas the area is 8.0 and 18.0 times larger than tiny classifiers, respectively.

Both tiny classifiers and XGBoost designs for blood and led are implemented with Pragmatic’s 0.8 μm FlexIC metal-oxide thin-film

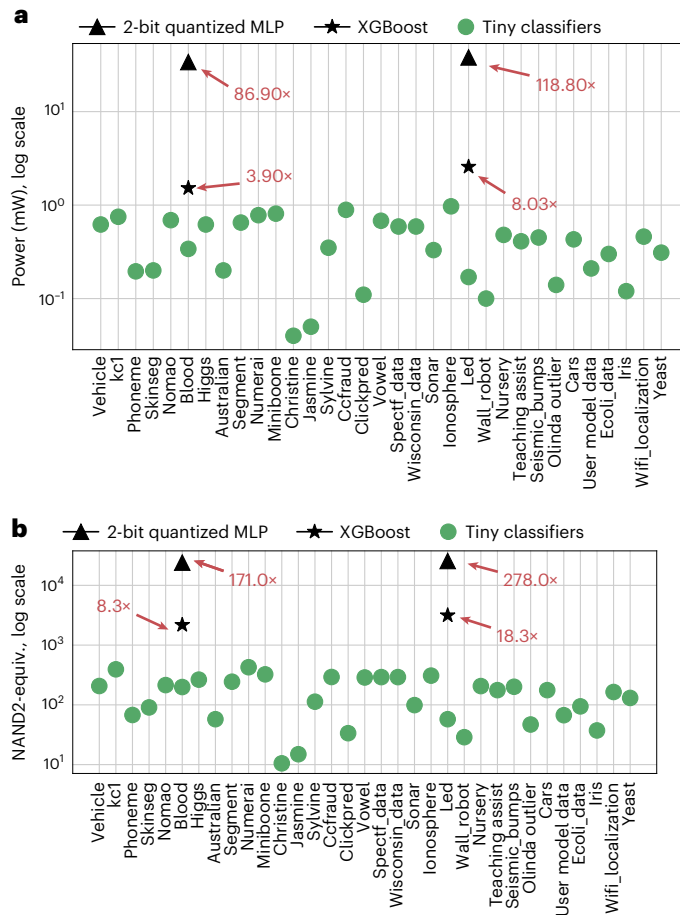


Fig. 6 | ASIC synthesis results of tiny classifiers and comparison against the two ML baselines. a, b, Power consumption (a) and NAND2-equivalent gate count (b) of tiny classifiers compared with MLP and AutoGluon XGBoost ASIC implementations for the blood and led datasets. The designs are synthesized using Synopsys Design Compiler targeting the open 45 nm PDK silicon technology.

transistor process in Pragmatic's FlexLogIC line⁴⁹. The designs are put through the Cadence implementation flow⁵⁰ to generate chip layouts.

Extended Data Fig. 2 shows the flexible chip layouts of the four designs. Extended Data 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, whereas it can run twice as fast. On the other hand, the comparative results for led are more prominent as tiny classifier is about 75 times smaller and consumes lower power as well as three times faster than XGBoost. An important observation is that the area variation of tiny classifiers between a binary and multiclass classification problem is negligible. Specifically, our methodology generates a smaller tiny classifier for led (105 NAND2-equivalent gates) compared with blood (150 NAND2-equivalent gates). In contrast, XGBoost implementation for led occupies five times more area than blood.

Tiny classifier and XGBoost designs for blood are fabricated as a FlexIC each on a 30- μ m-thick polyimide 200 mm wafer and tested. The tests of each FlexIC are undertaken in a wafer probe station. The input test vectors for the blood dataset are generated for each design through simulation, and the output signals from the simulation are used as the golden reference. Each input test vector is sent through the test probe card to the flexible chip under test, and the signals from the output pads are recorded and compared with the golden reference. Extended Data Fig. 3 shows the die photos of the tiny classifier FlexIC and the XGBoost FlexIC for the blood dataset. The waveforms (Extended Data Figs. 4 and 5) captured during the test show that the

test outputs match the golden reference (shown in blue colour) for numerous test vectors.

Our test results also show that the tiny classifier FlexICs have six times higher yield (that is, the ratio of the number of fully functional chips to the total number of fabricated chips) than the XGBoost FlexICs, which implies that the unit cost of a tiny classifier chip will be six times cheaper than that of XGBoost.

Conclusions

We have reported a methodology—termed auto tiny classifiers—to automatically generate classification circuits from tabular data. We identified a connection between graph-based genetic programming and classification problems in ML and developed an evolutionary approach to generate tiny classifier circuits composed of a small number of logic gates (less than 300 gates), which are capable of matching the performance of the state-of-the-art ML techniques. We evaluated the autogenerated tiny classifiers across datasets and reported the synthesis results and ML baselines designed in an ASIC in 45 nm silicon technology, showing improvements in area and power. We have also implemented tiny classifiers and XGBoost (smallest ML baseline) as flexible chips using the 0.8 μ m FlexIC thin-film transistor process technology. The full chip implementation results showed that the tiny classifiers could be clocked 2–3 times faster, were 10–75 times smaller and consumed lower power than XGBoost. The tiny classifiers are also six times cheaper to produce compared with XGBoost when fabricated as flexible chips.

Our methodology aims to generate classifier circuits for tabular data, but it is not—in principle—limited to tabular data. Work on recurrent-graph-based genetic programming⁵¹ indicates the general applicability of the evolutionary approach to other forms of data, such as time-series data. Our tiny classifiers could 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.

Data availability

The data that support the plots within this paper and other findings of this study are available from the corresponding author upon reasonable request.

Code availability

The code used to generate the plots within this paper is available from the corresponding author upon reasonable request.

References

- Arik, S. Ö. & Pfister, T. TabNet: attentive interpretable tabular learning. In *Proc. AAAI Conference on Artificial Intelligence* **35**, 6679–6687 (2021).
- Shwartz-Ziv, R. & Armon, A. Tabular data: deep learning is not all you need. *Inf. Fusion* **81**, 84–90 (2022).
- Popov, S., Morozov, S. & Babenko, A. Neural oblivious decision ensembles for deep learning on tabular data. Preprint at <https://doi.org/10.48550/arXiv.1909.06312> (2019).
- Kadra, A. et al. Well-tuned simple nets excel on tabular datasets. In *Proc. Neural Information Processing Systems* https://proceedings.neurips.cc/paper_files/paper/2021/file/c902b497eb972281fb5b4e206db38ee6-Paper.pdf (NIPS, 2021).
- Zhang, S., Yao, L., Sun, A. & Tay, Y. Deep learning based recommender system: a survey and new perspectives. *ACM Comput. Surv.* **52**, 5 (2019).
- Zhang, Y. et al. CADRE: Cloud-Assisted Drug REcommendation service for online pharmacies. *Mobile Netw. Appl.* **20**, 348–355 (2015).

7. Bao, Y. & Jiang, X. An intelligent medicine recommender system framework. In *2016 IEEE 11th Conference on Industrial Electronics and Applications (ICIEA)* 1383–1388 (IEEE, 2016).
8. Lee, J., Stanley, M., Spanias, A. & Tepedelenlioglu, C. Integrating machine learning in embedded sensor systems for Internet-of-Things applications. In *2016 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)* 290–294 (IEEE, 2016).
9. Ozer, E. et al. Binary neural network as a flexible integrated circuit for odour classification. In *2020 IEEE International Conference on Flexible and Printable Sensors and Systems (FLEPS)* 1–4 (IEEE, 2020).
10. Li, W., Logenthiran, T., Phan, V. & Woo, W. L. Implemented IoT-based Self-learning Home Management System (SHMS) for Singapore. *IEEE Internet Things J.* **5**, 2212–2219 (2018).
11. Kumar, PriyanMalarvizhi & Gandhi, UshaDevi A novel three-tier Internet of Things architecture with machine learning algorithm for early detection of heart diseases. *Comput. Elec. Eng.* **65**, 222–235 (2018).
12. Grinsztajn, L., Oyallon, E. & Varoquaux, G. Why do tree-based models still outperform deep learning on tabular data? Preprint at <https://doi.org/10.48550/arXiv.2207.08815> (2022).
13. Giraldo, J. S. P., Lauwereins, S., Badami, K. & Verhelst, M. Vocell: a 65-nm speech-triggered wake-up SoC for 10- μ W keyword spotting and speaker verification. *IEEE J. Solid-State Circuits* **55**, 868–878 (2020).
14. Mubarik, M. H. et al. Printed machine learning classifiers. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* 73–87 (IEEE, 2020).
15. Weller, D. D. et al. Printed stochastic computing neural networks. In *2021 Design, Automation and Test in Europe Conference and Exhibition (DATE)* 914–919 (IEEE, 2021).
16. Biggs, J. et al. A natively flexible 32-bit Arm microprocessor. *Nature* **595**, 532–536 (2021).
17. Bleier, N. et al. FlexiCores: low footprint, high yield, field reprogrammable flexible microprocessors. In *Proc. 49th Annual International Symposium on Computer Architecture (ISCA '22)* 831–846 (ACM, 2022).
18. Ozer, E. et al. Bespoke machine learning processor development framework on flexible substrates. In *2019 IEEE International Conference on Flexible and Printable Sensors and Systems (FLEPS)* 1–3 (IEEE, 2019).
19. Ozer, E. et al. A hardwired machine learning processing engine fabricated with submicron metal-oxide thin-film transistors on a flexible substrate. *Nat. Electron.* **3**, 419–425 (2020).
20. Ozer, E. et al. Malodour classification with low-cost flexible electronics. *Nat. Commun.* **14**, 777 (2023).
21. Zhou, F. & Chai, Y. Near-sensor and in-sensor computing. *Nat. Electron.* **3**, 664–671 (2020).
22. Iyer, R. & Ozer, E. Visual IoT: architectural challenges and opportunities; toward a self-learning and energy-neutral IoT. *IEEE Micro.* **36**, 45–49 (2016).
23. Miller, J. F. & Harding, S. L. Cartesian genetic programming. In *Proc. 10th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '08)* 2701–2726 (ACM, 2008).
24. Atkinson, T., Plump, D. & Stepney, S. Evolving graphs by graph programming. In *European Conference on Genetic Programming* **10781** (Springer, 2018).
25. Brameier, M. F. and Banzhaf, W. *Linear Genetic Programming* (Springer Science & Business Media, 2007).
26. Poli, R. et al. Evolution of graph-like programs with parallel distributed genetic programming. in *ICGA* 346–353 (Citeseer, 1997).
27. Leitner, J., Harding, S., Forster, A. & Schmidhuber, J. Mars terrain image classification using Cartesian genetic programming. In *Proc. 11th International Symposium on Artificial Intelligence, Robotics and Automation in Space, i-SAIRAS* 1–8 (European Space Agency, 2012).
28. Parziale, A., Senatore, R., Della Cioppa, A. & Marcelli, A. Cartesian genetic programming for diagnosis of Parkinson disease through handwriting analysis: performance vs. interpretability issues. *Artif. Intell. Med.* **111**, 101984 (2021).
29. Brameier, M. & Banzhaf, W. Evolving teams of predictors with linear genetic programming. *Genet. Program. Evolvable Mach.* **2**, 381–407 (2001).
30. Miller, J. F., Thomson, P. & Fogarty, T. Designing electronic circuits using evolutionary algorithms. Arithmetic circuits: a case study. in *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science* 105–131 (Wiley, 1997).
31. Miller, J. F. et al. An empirical study of the efficiency of learning Boolean functions using a Cartesian genetic programming approach. In *Proc. 1st Annual Conference on Genetic and Evolutionary Computation* **2**, 1135–1142 (1999).
32. Sotto, L. F. D., Kaufmann, P., Atkinson, T., Kalkreuth, R. & Basgalupp, M. P. A study on graph representations for genetic programming. In *Proc. 2020 Genetic and Evolutionary Computation Conference* 931–939 (ACM, 2020).
33. Franoso Dal Piccol Sotto, L., Kaufmann, P., Atkinson, T., Kalkreuth, R. & Porto Basgalupp, M. Graph representations in genetic programming. *Genet. Program. Evolvable Mach.* **22**, 607–636 (2021).
34. Walker, J. A. & Miller, J. F. The automatic acquisition, evolution and reuse of modules in Cartesian genetic programming. *IEEE Trans. Evol. Comput.* **12**, 397–417 (2008).
35. Harding, S. L., Miller, J. F. & Banzhaf, W. Self-modifying Cartesian genetic programming. in *Cartesian Genetic Programming* 101–124 (Springer, 2011).
36. Hodan, D., Mrazek, V. & Vasicek, Z. Semantically-oriented mutation operator in Cartesian genetic programming for evolutionary circuit design. *Genet. Program. Evolvable Mach.* **22**, 539–572 (2021).
37. Vasicek, Z. & Sekanina, L. Evolutionary approach to approximate digital circuits design. *IEEE Trans. Evol. Comput.* **19**, 432–444 (2014).
38. Mrazek, V., Hrbacek, R., Vasicek, Z. & Sekanina, L. Evoapprox8b: library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)* 258–261 (IEEE, 2017).
39. Chen, T. & Guestrin, C. XGBoost: a scalable tree boosting system. In *Proc. 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* 785–794 (ACM, 2016).
40. Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A. V. & Gulin, A. CatBoost: unbiased boosting with categorical features. *Adv. Neural Inf. Proc. Sys.* **31** (2018).
41. Miller, J. F. Cartesian genetic programming: its status and future. *Genet. Program. Evolvable Mach.* **21**, 129–168 (2020).
42. Kimura, M. *The Neutral Theory of Molecular Evolution* (Cambridge Univ. Press, 1983).
43. Atkinson, T., Plump, D. & Stepney, S. Evolving graphs with semantic neutral drift. *Nat. Comput.* **20**, 127–143 (2021).
44. Vanschoren, J., van Rijn, J. N., Bischl, B. & Torgo, L. OpenML: networked science in machine learning. *SIGKDD Explor. Newsl.* **15**, 49–60 (2013).
45. Dua, D. & Graff, C. UCI machine learning repository; <https://archive.ics.uci.edu/> (2017).
46. Kaggle. <https://www.kaggle.com>
47. Erickson, N. et al. AutoGluon-Tabular: robust and accurate AutoML for structured data. Preprint at <https://doi.org/10.48550/arXiv.2003.06505> (2020).
48. FreePDK45. Standard Cell Library 45nm.

49. FlexLogIC; <https://www.pragmaticsemi.com/create-more/devices> (2022).
50. Cadence Innovus Implementation System. https://www.cadence.com/en_US/home/resources/datasheets/innovus-implementation-system-ds.html (2024).
51. Atkinson, T. *Evolving Graphs by Graph Programming*. PhD thesis, Univ. of York (2019).

Acknowledgements

K.I. is funded by an Arm Ltd and EPSRC iCASE PhD Scholarship. M.L. is funded by an Arm/RAEng Research Chair award and a Royal Society Wolfson Fellowship. The research carried out by T.A. happened while being an employee of the University of Manchester. The research is partially funded by EPSRC LAMBDA (EP/N035127/1), EnnCore (EP/T026995/1) and UKRI NimbleAI (no. 10039070).

Author contributions

G.B., E.O., J.K. and J.B. conceived the concept of evolving a graph of gates as an ML classifier for tabular data. K.I. and M.L. developed the methodology and software toolflow for generating tiny classifier circuits using the graph-based genetic programming algorithm developed by T.A. All the authors helped with the design of the experiments. K.I. implemented and conducted the experiments and designed the tiny classifiers and XGBoost classification hardware blocks. J.K. performed the flexible chip implementation of the tiny classifiers and XGBoost. J.K. and G.A. tested and evaluated the fabricated FlexICs. Finally, K.I., E.O., T.A. and M.L. wrote the Article.

Competing interests

The authors declare no competing interests.

Additional information

Extended data is available for this paper at <https://doi.org/10.1038/s41928-024-01157-5>.

Correspondence and requests for materials should be addressed to Konstantinos Iordanou.

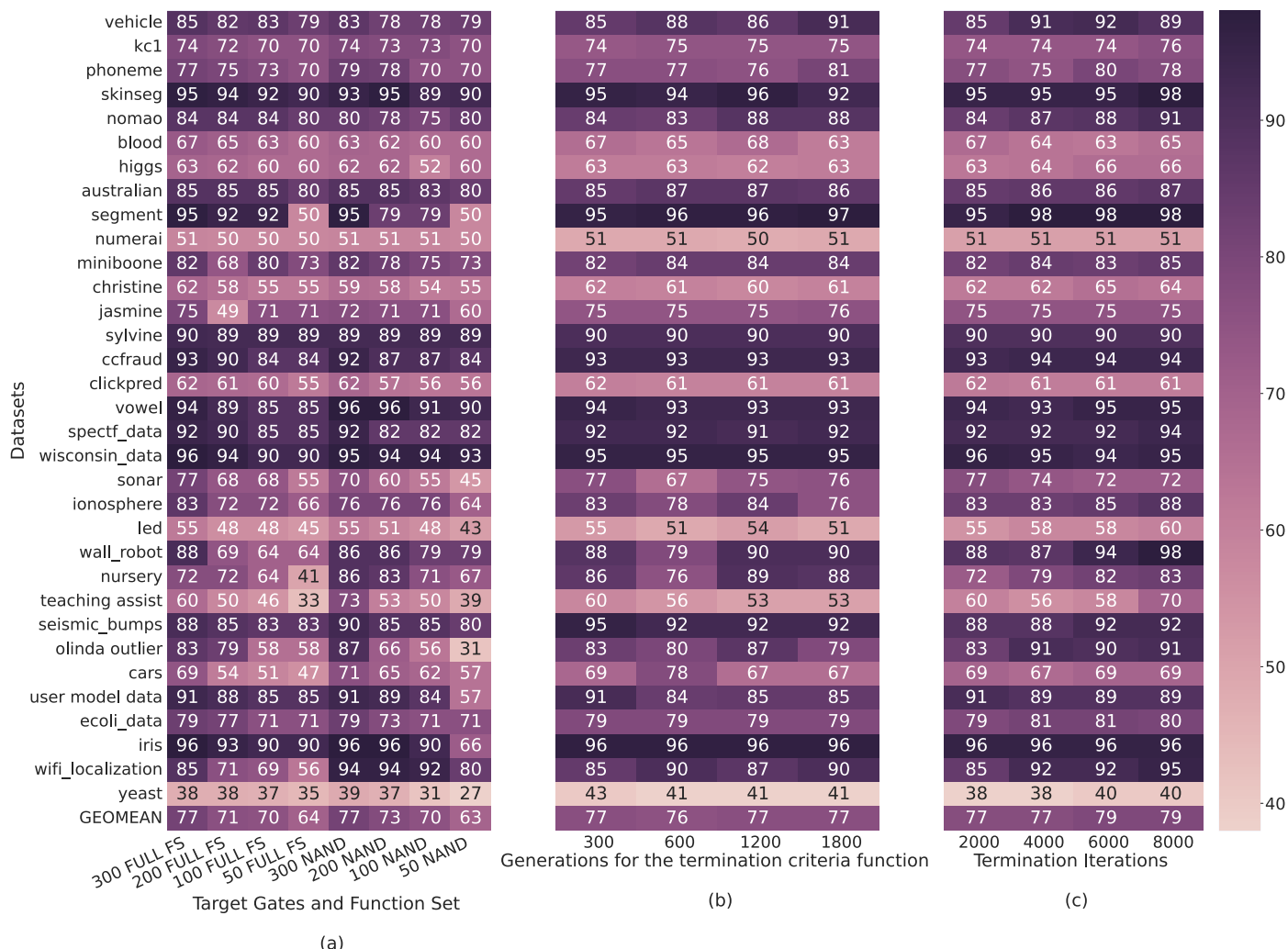
Peer review information *Nature Electronics* thanks Pallab Datta, Chris Winstead and the other, anonymous, reviewer(s) for their contribution to the peer review of this work.

Reprints and permissions information is available at www.nature.com/reprints.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

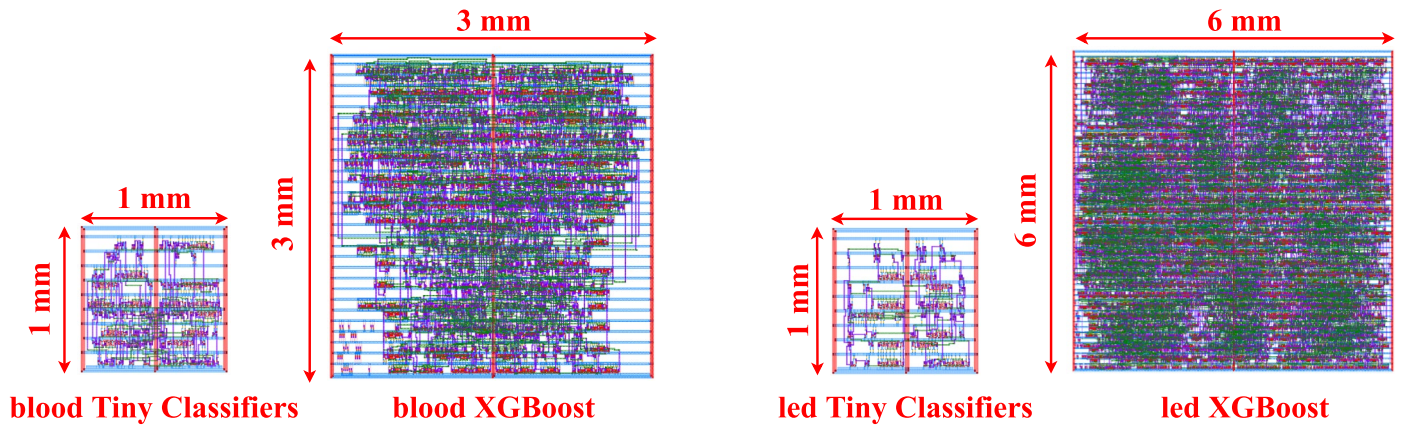
Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

© The Author(s) 2024

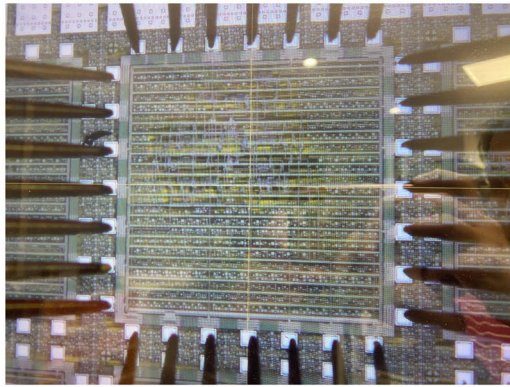


Extended Data Fig. 1 | Prediction Accuracy Analysis of Tiny Classifiers with hyper-parameter tuning. (a) Accuracy vs. number of gates. Generations for the termination function is 300 and termination iterations is 2000. *FullFS* indicates that the generated circuit will be constructed with logical gates within the function set $F = \{\text{and, or, nand, nor}\}$. For NAND function set the generated

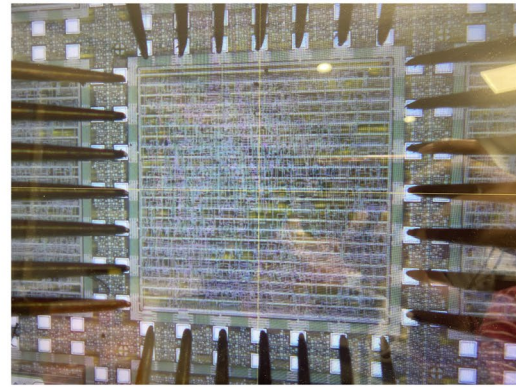
circuits constructed only with NAND gates. (b) Accuracy vs. generations for the termination function. The number of gates is 300 and the number of termination iterations is 2000. (c) Accuracy vs. the number of termination iterations. The number of gates and the generations for the termination function are both set to 300.



Extended Data Fig. 2 | Flexible chip layouts of Tiny Classifiers and XGBoost. The flexible chips are implemented in Pragmatic's $0.8\mu\text{m}$ FlexIC TFT process for blood and led datasets.

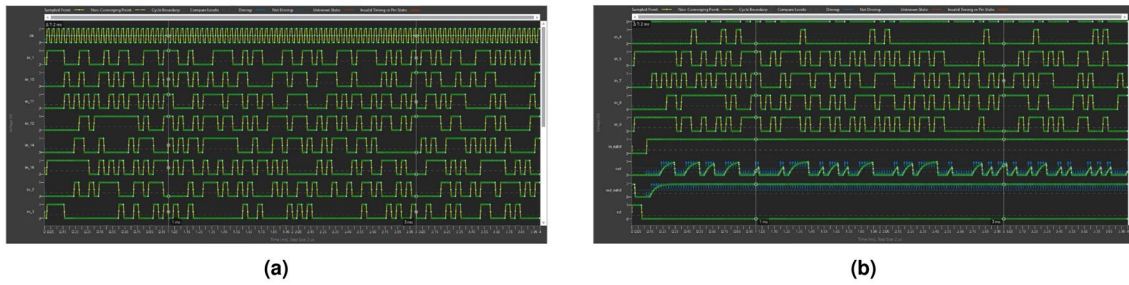


(a) Tiny Classifier FlexIC.

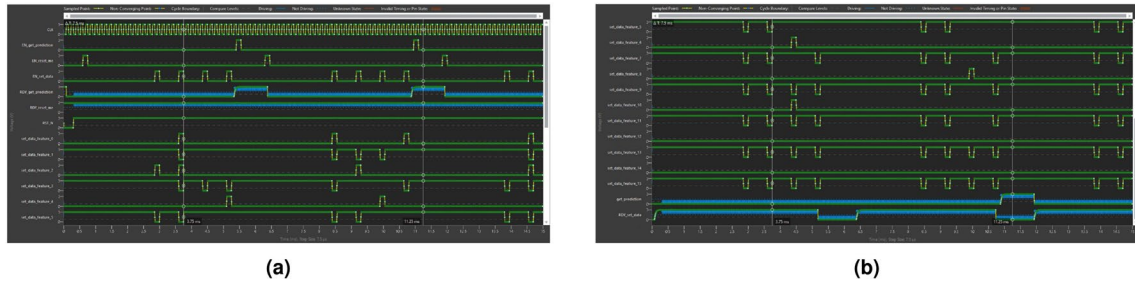


(b) XGBoost FlexIC.

Extended Data Fig. 3 | Die photos of Tiny Classifier (a) and XGBoost (b) FlexICs for *blood* dataset. Tiny Classifier and XGBoost designs for blood are fabricated as a FlexIC each on a 30 μ m thick polyimide 200mm wafer and tested. The tests of each FlexIC are undertaken in a wafer probe station.



Extended Data Fig. 4 | Waveforms capture the test outputs on a specific number of input-test vectors for the Tiny Classifier FlexIC. a,b, The test outputs were compared to the expected output shown in blue.



Extended Data Fig. 5 | Waveforms capture the test outputs on a specific number of input-test vectors for the XGBoost FlexIC. a,b, The test outputs were compared to the expected output shown in blue.

Extended Data Table 1 | The dataset collection

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
wif -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
†numera1 (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

The experiments use a comprehensive collection of 33 tabular datasets, mainly from OpenML, UCI and Kaggle. Note: † indicates that the dataset is part of the evaluation of AutoGluon⁴⁷.

Extended Data Table 2 | Tiny Classifiers and XGBoost implementation results

	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

Tiny Classifiers and XGBoost designs for blood and led are implemented with Pragmatic's 0.8 μ m FlexIC metal-oxide thin-film transistor (TFT) process in Pragmatic's FlexLogIC line at 3V supply voltage.