electronics

MDPI

Article

Fast and Energy-Efficient Oblique Decision Tree Implementation with Potential Error Detection

Chungsoo Lim

Department of Electronic Engineering, Korea National University of Transportation, Cheongju-si 27469, Republic of Korea; clim@ut.ac.kr

Abstract: In the contemporary landscape, with the proliferation of cyber-physical systems and the Internet of Things, intelligent embedded systems have become ubiquitous. These systems derive their intelligence from machine learning algorithms that are integrated within them. Among many machine learning algorithms, decision trees are often favored for implementation in such systems due to their simplicity and commendable classification performance. In this regard, we have proposed the efficient implementations of a fixed-point decision tree tailored for embedded systems. The proposed approach begins by identifying an input vector that might be classified differently by a fixed-point decision tree than by a floating-point decision tree. Upon identification, an error flag is activated, signaling a potential misclassification. This flag serves to bypass or disable the subsequent classification procedures for the identified input vector, thereby conserving energy and reducing classification latency. Subsequently, the input vector is alternatively classified based on class probabilities gathered during the training phase. In comparison with traditional fixed-point implementations, our proposed approach is proven to be 23.9% faster in terms of classification speed, consuming 11.5% less energy without compromising classification accuracy. The proposed implementation, if adopted in a smart embedded device, can provide a more responsive service to its users as well as longer battery life.

Keywords: decision tree; fixed-point arithmetic; hardware implementation

Citation: Lim, C. Fast and Energy-Efficient Oblique Decision Tree Implementation with Potential Error Detection. *Electronics* **2024**, *13*, 410. https://doi.org/10.3390/electronics13020410

Academic Editor: Andrea Bonci

Received: 17 November 2023
Revised: 10 January 2024
Accepted: 15 January 2024
Published: 18 January 2024

1. Introduction

In the era of artificial intelligence, the significance of machine learning algorithms remains paramount, playing a crucial role in a myriad of applications across various domains [1–4]. A decision tree (DT) is an effective classification algorithm proposed by Breiman et al. [5]. It has been utilized in diverse applications such as handoffs in Internet of Vehicles [6], screen content coding [7], human activity classification [8], wafer map failure pattern recognition [9], forecasting consumer decision [10], and selecting pet adopters [11]. A DT can be implemented either in software or in hardware, and hardware implementation has recently drawn considerable attention [12,13]. There are two primary reasons for this shift. Firstly, the current technological trend strongly advocates for the hardware implementation of machine learning algorithms. With the maturation of technologies such as cyber-physical systems and the Internet of Things, data-driven classifications or recognition using machine learning algorithms have become ubiquitous. As the volume of available data and the number of users continue to grow exponentially, running machine learning algorithms in a centralized manner is no longer a scalable option. Consequently, a distributed computing framework like edge computing, which situates computation and data storage closer to the data sources, has been developed to enhance response time and conserve network bandwidth. This trend underscores the increasing likelihood of implementing machine learning algorithms in computer systems, where energy efficiency is a paramount design consideration. Therefore, the hardware implementation of machine learning algorithms is deemed a viable and relevant option. Secondly, among the multitude

of machine learning algorithms, the DT algorithm stands out as a leading candidate for hardware implementation. This is attributed to the relative simplicity of the DT algorithm coupled with its ability to deliver acceptable performance [14]. Its simplicity can be translated to low energy consumption and fast response time. Examples of DT hardware implementation in real-time embedded systems include cow behavior classification [13], biomedical monitoring systems [15], network traffic classifiers [16], gesture recognition systems [17], and gas identification [18].

Between two types of DTs, axis-parallel DTs, whose hyperplanes are parallel to the axis in attribute space, have been more popular than oblique DTs, whose hyperplanes are oblique to the axis. However, if data can be more readily partitioned by non-axis-parallel hyperplanes, oblique DTs are often simpler but more accurate [19]. This difference may be more apparently manifested as data size and complexity grow in the big data era. Therefore, oblique DTs have gained interest and have been studied extensively. Some of the research works include the tree induction algorithm [19] and oblique DT ensemble [20].

Some DT hardware implementations for embedded systems adopt fixed-point arithmetic to reduce hardware requirements and energy consumption [21,22]. Compared with floating-point arithmetic, however, fixed-point arithmetic may suffer from lower classification accuracy due to its limited capability of representing real numbers. Hence, the selection of the number of bits to represent real values in fixed-point arithmetic hardware becomes a critical consideration. It must be carefully chosen to strike a balance between maintaining classification accuracy and maximizing resource efficiency. This trade-off may entail a certain degree of accuracy decline, particularly in systems where energy consumption is the most crucial design consideration. In such instances, identifying which classification outputs are likely to differ from those produced by floating-point arithmetic allows for the possibility of disregarding those identified outputs, potentially enhancing the precision of classifications. In such a case, if it becomes feasible to determine whether an input vector is prone to being classified differently by a fixed-point DT than by a floating-point DT, the processing of the input vector can be halted for improved classification speed and energy efficiency. Alternatively, the corresponding classification can be ignored, leading to higher precision.

In this context, we propose fixed-point oblique DT implementations for embedded systems, aiming at improving classification speed and energy efficiency without degrading classification accuracy. The proposed implementations encompass three key mechanisms. Firstly, a potential error detection mechanism identifies an input vector for which fixed-point arithmetic and floating-point arithmetic may yield different classification outputs. Subsequently, a potential error flag is set for the input vector, and this error flag is included in the final outputs along with the predicted class identification number, indicating that the confidence level of the classification is low. The second mechanism, termed the skipping/disabling mechanism, examines the error flag to determine whether to continue processing the corresponding input vector or not. If the input vector is flagged as potentially erroneous, the mechanism makes the rest of the classification procedures of the input vector skipped or disabled, enhancing energy efficiency and reducing classification latency. The third mechanism predicts the class of a skipped or disabled input vector. This step is essential for the skipped or disabled input vectors as their classification outputs were interrupted before being produced.

The novelty of the proposed implementation can be encapsulated in two key points. Firstly, it pioneers the utilization of errors caused by the limitation of fixed-point representation, opening up an opportunity for faster and energy-efficient DT classification. Secondly, the implementation features an algorithm that generates an alternative classification result prior to reaching a leaf node.

The rest of the paper is organized as follows: Section 2 presents the literature review, and Section 3 introduces the baseline DT architectures. Section 4 describes the proposed DT architectures, and Section 5 is devoted to experimental results; the paper is concluded in Section 6.

2. Related Works

There have been quite a few hardware implementations of DTs aiming to achieve higher performance and energy efficiency than software implementations. A highly scalable and parallelized axis-parallel DT implementation was proposed in [23]. This pipelined implementation achieves high levels of parallelism by increasing the number of parallel pipelines, and it can accommodate trees with depths of up to 13 by increasing the number of pipeline stages. Another scalable approach targeting large DTs was proposed in [24]. The authors pinpointed the high memory requirement of a large DT as the most critical hindrance to scalability. To solve this problem, they first partitioned tree data and stored it in dual-port distributed RAM in each individual processing element, which constituted a two-dimensional pipelined architecture.

An implementation for oblique DTs was also proposed [25]. Because an oblique DT is a generalized version of an axis-parallel DT, this implementation can cover both axis-parallel and oblique DTs. They presented two architectures: Single Module per Level architecture and universal node architecture. In the Single Module per Level architecture, each module is connected in a pipeline fashion, processing multiple inputs simultaneously. The universal node architecture is suitable for DT ensembles due to its architectural simplicity.

A recent paper introduced a template-based architecture for shallow machine learning algorithms, such as support vector machines, logistic regressions, k-nearest neighbors, and DTs [26]. They targeted shallow machine learning algorithms because they can complement deep learning algorithms with acceptable accuracy, especially in edge computing. They proposed a methodology that automatically builds accelerator hardware for a machine learning algorithm using templates that match the algorithm's computational structure.

Implementations for DT ensembles have also been one of the hot topics in machine learning hardware research because they are very effective in the regime of limited training data, little training time, and little experience for parameter tuning. Also, in some applications such as pixel classification of hyperspectral images, DT ensembles produce an accuracy close to that obtained with convolutional neural networks while executing one order of magnitude less computation [14]. The main difference between DT implementations and DT ensemble implementations is the existence of a combiner module that combines the outputs of individual ensemble members to generate a collective decision. Implementations of axis-parallel DT ensembles were proposed in [27,28], and those of oblique DT ensembles can be found in [29]. These implementations targeted real-time embedded systems with stringent power constraints.

An example of another line of research is an implementation of a state-of-the-art tree traversal algorithm [30]. They present a system-on-chip (SoC)-based field-programmable gate arrays (FPGAs) implementation of the QuickScorer algorithm [31], which is an efficient tree traversal algorithm designed for large binary tree ensembles. This solution is suitable for difficult inference tasks such as ranking documents, web search engines, and online social networks.

Broadly speaking, there have been two predominant research directions in DT hardware implementation. The first direction is a hardware-centric approach, which involves techniques such as parallelization [23] and memory system optimization [24]. The focus of this approach is on enhancing the hardware design of existing algorithms to boost performance and energy efficiency. The second direction, on the other hand, adopts efficient, hardware-friendly algorithms for actual implementation [31]. What we propose in this paper diverges from these two directions through its leveraging of the algorithmic features inherent in DTs. The first algorithmic feature we utilize is the computation at each node of an oblique DT: the sum of products between the input vector and node coefficients. This computational feature, combined with fixed-point arithmetic, contributes to potential error detection, as detailed in Section 4.1. The second algorithmic feature we pay attention to is the tree-based nature of DT algorithms. As tree traversal progresses down a tree for an input vector, it becomes increasingly probable to predict the class of the input vector correctly, even before reaching a leaf node. This is because the number of candidate

leaf nodes at which the tree traversal will conclude is reduced. This feature is harnessed to predict the classes of input vectors detected as potentially erroneous, as explained in Section 4.3. Because these features are common in oblique DTs, our proposed approach can seamlessly integrate with other hardware-centric implementation techniques developed for oblique DTs.

3. Baseline DT Architectures

At each node of an oblique DT, the following equation is evaluated to choose the next node to visit.

$$f(\boldsymbol{A}) = \sum_{i=1}^{n} a_i A_i + a_{n+1} \lessgtr 0 \tag{1}$$

where A_i denotes the i^{th} attribute of input vector $\boldsymbol{A}$ (length of n), a_i denotes the i^{th} coefficient of a node except with a_{n+1} being the bias term, and 0 is the threshold. If $f(\boldsymbol{A})$ is greater than or equal to the threshold, the left child node is selected as the next node to visit. Otherwise, the right child node is selected. The proposed DT architectures implement Equation (1) and are based on two existing DT architectures: pipeline architecture and recurrent architecture [25]. As shown in Figure 1, the pipeline architecture is composed of multiple pipeline stages, each of which is associated with a DT level. On the other hand, the recurrent architecture depicted in Figure 2 has one universal stage, and an input vector stays in the stage until it is classified.

A stage in the pipeline architecture comprises four modules: a control module (*CTL*), a node memory, an input memory, and a computation module (*COMP*). A control module broadcasts a state number generated from a state machine in the module to all other modules in the same stage in order to orchestrate each module's execution. A control module is also responsible for generating a disable signal. In addition, a control module selects a node number and a class number for the next stage and delivers them to the next stage. A node memory module consists of three memory sub-modules: a coefficient memory, a node number memory, and a class number memory. The coefficient memory stores coefficients for all nodes at the same level, while the node number memory and class number memory store child node numbers and their corresponding class numbers, respectively. For a leaf child node, the stored class number is valid, whereas for a non-leaf child node, it is an invalid class number that is preset to be larger than any valid class number. When receiving a node number from the previous stage, these memory sub-modules provide the control module with node coefficients, a class number, and two child node numbers based on the received node number. An input memory in the pipeline architecture holds a single input vector, which is then relayed to the next stage after being processed in the current stage. The computation module performs calculations using Equation (1), utilizing the input vector and a coefficient vector from the input memory and the coefficient memory within the same stage. It is important to note that solid lines and dashed lines are used to indicate data flows and control signal flows, respectively. The control signal conveys the current state number generated by the state machine in the control module. This comprehensive architecture facilitates the efficient processing and coordination of information throughout the pipeline.

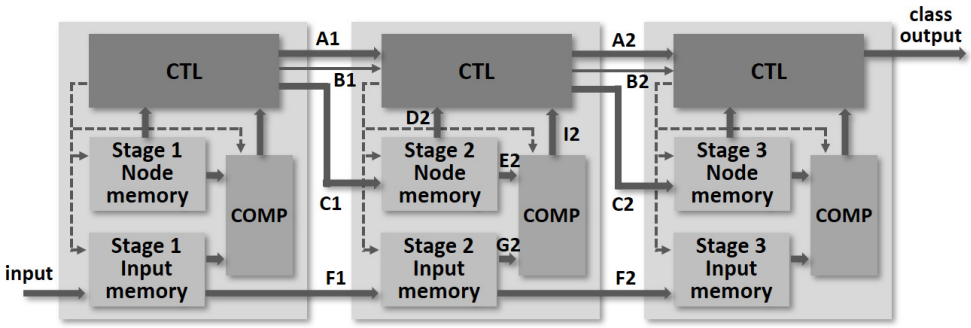

Figure 1. Baseline decision tree (DT) architecture 1: pipelined DT with three pipeline stages.

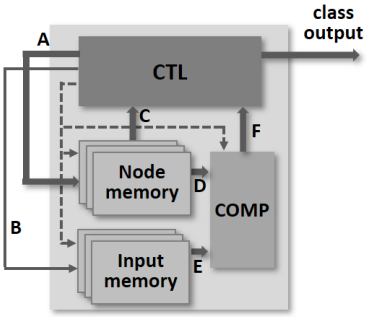

Figure 2. Baseline DT architecture 2: recurrent DT (A: class number, B: disable signal, C: node number, D: child node number and their class number, E: node coefficients, F: input vector).

Let us explain how the pipeline architecture works using Figure 1. The control module in pipeline stage 2 receives a class number (*A1*) and the disable signal (*B1*) from stage 1. If a leaf node is reached for an input vector in stage 1, the disable signal (*B1*) is true and the class number (*A1*) is a valid one. In this case, the class number is relayed to stage 3 and is outputted as the final output for the input vector. The purpose of the disable signal is to prevent the remaining pipeline stages (stage 2 and 3) from unnecessarily processing the input vector. In order to achieve this, the control module generates different state numbers if the disable signal is true. The node memory in stage 2 receives a node number (*C1*) from stage 1, which corresponds to the node that will be processed in stage 2. Upon receiving the node number, the node memory provides data that correspond to the node number: the child node numbers and their class numbers (*D2*) in the control module and the node coefficients (*E2*) in the computation module. The input memory in stage 2 receives an input vector (*F1*) from stage 1, which has been processed in stage 1. The input memory then sends the input vector to the computation module (*G2*) as well as to the input memory in stage 3 (*F2*). The result computed by the computation module (*I2*) is delivered to the control module, and the control module selects a child node based on the result from the computation module and sends it to stage 3 (*C2*). If a leaf node is reached in stage 2, the disable signal (*B2*) becomes true, and the corresponding class number (*A2*) is passed to stage 3, which is chosen between the class numbers from the node memory (*D2*).

The recurrent architecture comprises a singular stage, mirroring the structure of a stage in the pipeline architecture as in Figure 2. Within the recurrent architecture, nodes that traverse during the classification of an input vector are sequentially processed within this single stage. Unlike the pipeline architecture, where each stage's input memory module stores only one vector, the input memory module in the recurrent architecture stores multiple input vectors. A memory counter within the input memory module is employed to select an input vector for output, incrementing by one each time an input vector is classified. In contrast to the node memory in the pipeline architecture, which stores data for a specific tree level, the node memory in the recurrent architecture encompasses data for all tree levels. The computation module in the recurrent architecture mirrors the one employed in the pipeline architecture. Notably, in the recurrent architecture, the control

module transmits the next node number to the node memory within the same stage instead of forwarding it to the node memory in the subsequent stage. Additionally, the control module generates a skip signal instead of a disable signal. This skip signal instructs the input memory to output the next input vector, streamlining the recurrent architecture's processing of input vectors. The sequential nature of node processing and the ability to store and retrieve data across all tree levels contribute to the distinctive characteristics of the recurrent architecture.

As shown in Figure 2, the recurrent architecture begins to work by sending the initial node number (*A*) to the node memory and setting the skip signal (*B*) to true, which is connected to the input memory. Then, the node memory sends corresponding child node numbers and class numbers (*C*) to the control module and corresponding node coefficients (*D*) to the computation module. At the same time, the input memory sends the first input vector (*E*) to the computation module. The computation module uses node coefficients (*D*) and an input vector (*E*) to produce the output of Equation (1) and sends the output (*F*) to the control module. The control module then examines the output value (*F*) to decide which child node to visit next. When the next node is selected, the control module uses class numbers to determine whether a leaf node has been reached. If the class number of the selected child node is invalid, it means the next node is not a leaf node. In this case, the next node number is selected between the child node numbers from the node memory and is sent to the node memory. If the next node turns out to be a leaf node—in other words, if the class number of the selected child node is a valid class number—the control module outputs the class number as the final classification output and sets the skip signal to true in order to start processing the next input vector. More detailed information on the baseline architectures can be found in [25].

4. Proposed DT Architectures

We propose two DT architectures: a pipeline architecture and a recurrent architecture. To enhance execution time and energy efficiency, we introduce the following novel features to the baseline DT architectures. Firstly, our proposed architectures can detect potential misclassifications resulting from the limitations of fixed-point representation. This detection mechanism serves to identify input vectors that may be prone to misclassification due to the finite precision of fixed-point representation. Secondly, by leveraging this detection, we optimize execution time and energy consumption by halting tree traversal for the identified input vectors. The rationale behind this mechanism is rooted in the understanding that continuing the processing of a detected input vector that is likely to be misclassified is meaningless. If an input vector is destined for misclassification, its class may even be randomly generated. Instead of making a random classification, our approach predicts the class number for the detected input vector. During the DT training phase, we learn the most frequently observed class of training vectors at each node, and this learned class is then utilized as the classification for input vectors identified as potentially misclassified at that particular node. In the subsequent subsections, we will elaborate on these three distinctive features.

4.1. Potential Error Detection

Approximating a real number using a fixed-point binary number involves rounding. When rounding up is employed to approximate the multiplication outputs during the evaluation of Equation (1), the final result of the equation tends to be greater than or equal to its true value. Conversely, if rounding down is used, the result of the equation is less than or equal to its true value. This approximation error, stemming from rounding, can sometimes lead to different child node selections compared with scenarios where floating-point multiplications are utilized. Furthermore, the occurrence of one or more distinct child node selections during the processing of an input vector may result in a switch in classification result. In essence, the approximation errors introduced by rounding can influence the decision-making process in the tree traversal, potentially leading to variations

in the selected child nodes and subsequently affecting the final classification outcome for the input vector.

Table 1 shows the diversion ratio and class switch ratio of each dataset selected from the UCI machine learning database repository [32]. Note that the fixed-point bit length used to generate Table 1 is 16: 11 bits for the integer part and 5 bits for the fractional part. The *diversion ratio* column contains ratios between the number of input vectors for which one or more diversions occur and the total number of input vectors. It should be noted that a diversion here means a different child node selection. A class switch ratio indicates what the percentage of the number of input vectors with classification switches is with respect to the number of input vectors for which one or more diversions occur. On average, 7.3% of input vectors undergo diversions, and 70.6% of them suffer from classification switches. For *Car*, *Iris*, and *Scale*, the impact of the class switch seems negligible, but it is not ignorable for *Ecoli*, *Diabetes*, *Glass*, and *Liver*.

Given the challenge of precisely identifying cases where rounding-induced errors occur, the typical solution has been to increase the fixed-point bit length to uphold classification accuracy. Unfortunately, this strategy comes at the cost of forfeiting opportunities for achieving higher energy efficiency and faster classification speeds. However, we contend that there is an alternative approach. If we have access to the outputs of Equation (1), computed using each of the two rounding schemes for a given input vector, we can discern whether the corresponding child node selection is potentially erroneous. This insight not only allows for a more nuanced understanding of potential misclassifications but also opens the door to opportunities for enhancing both energy efficiency and classification speed. By leveraging this information, it becomes possible to make informed decisions about when to intervene in the processing of an input vector.

Table 1. Diversion ratio and class switch ratio of selected datasets from [32] when using a 16-bit fixed-point representation.

Dataset	Diversion Ratio	Class Switch Ratio
Car	0.009	0.525
Ecoli	0.109	0.791
Wine	0.034	0.917
Survival	0.081	0.512
Diabetes	0.142	0.618
Iris	0.018	0.833
Hayes	0.040	0.667
Scale	0.011	0.750
Ion	0.042	0.628
Glass	0.081	0.911
Haber	0.081	0.681
Liver	0.174	0.711
Raisin	0.113	0.549
Average	0.073	0.706

Figure 3 shows three outputs of Equation (1): an output with rounding down (f^{down}), an output without rounding (f^{orig}), and an output with rounding up (f^{up}). If the threshold (i.e., zero) is in *range 1* or *range 4*, the next nodes determined by the three outputs are identical, which means that fixed-point computation does not alter the next node to visit. In contrast, if the threshold is in *range 2* or *range 3*, one of the child nodes determined using rounding is different from the child node determined without rounding, and the child node determined using rounding down differs from that using rounding up. Hence, if the

output of Equation (1) using rounding up and that using rounding down have different signs, it indicates that there is a chance of rounding-induced diversion and classification switch for the corresponding input vector, as presented in Table 1.

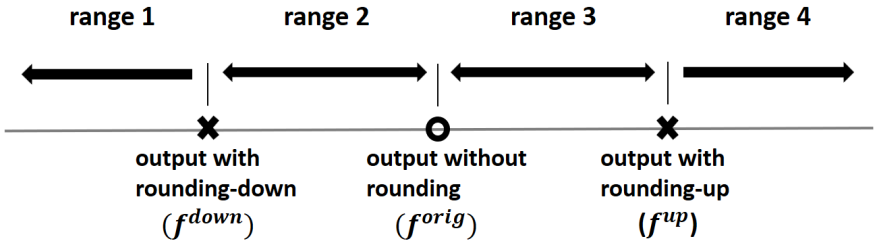

Figure 3. Output of Equation (1) with rounding down, rounding up, and no rounding.

However, in order to make use of this observation, Equation (1) should be evaluated twice for a node using rounding up and rounding down separately. To cope with this inefficiency, we propose a more efficient scheme based on another observation: the difference between the rounded-up version of a value and the rounded-down version is at most the smallest positive value (SPV) of the corresponding fixed-point format (one at the least significant bit and zero at all other bits). Because each multiplication in Equation (1) entails a rounding operation, the difference between f^{up} and f^{down} at each node is at most n×SPV, where n is the number of multiplications in Equation (1). This observation allows us to approximate f^{up} by adding n×SPV to f^{down} or f^{down} by subtracting n×SPV from f^{up}. Because n×SPV is a constant for a given dataset, it can be stored in a register instead of computing it. If this method detects an erroneous child node selection, which potentially gives rise to a classification error, an error flag is set. This flag not only serves as classification confidence but also gets utilized later for faster and more energy-efficient classification.

As shown in Figure 4, implementing this mechanism requires three additional components: an adder to approximate f^{up} from f^{down}, a register that stores n×SPV, and a comparator that compares the most significant bits of f^{up} and f^{down}. The exclusive-OR gate can be used as the comparator to produce the error flag, which becomes one when the two most significant bits are different. This indicates a possible diversion in child node selection.

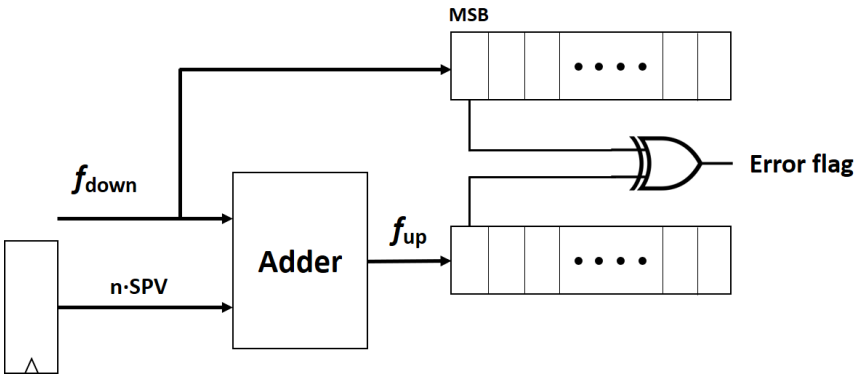

Figure 4. Extra hardware for potential error detection.

4.2. Two Hardware Schemes Utilizing Error Detection

If fixed-point arithmetic and floating-point arithmetic produce different classification outputs, at least one of them is inevitably incorrect. If the overall classification accuracy with floating-point arithmetic is high, it is highly probable that the classification output using fixed-point arithmetic is incorrect. Thus, there has to be a way to stop processing

an input vector once the potential error flag is set for the vector. For this, we propose two schemes: disabling and skipping schemes.

The proposed disabling and skipping schemes suit the pipeline and the recurrent architectures, respectively. In the pipeline architecture, if the potential error flag is set for an input vector, the disable signal is set and relayed to all remaining stages. The signal disables the memory accesses to the two memory modules in all remaining stages and makes the inputs to the arithmetic units unchanged, thereby abating switching activity. In addition, the error flag is also outputted along with classification output to indicate a confidence level of the classification output. As the baseline pipeline architecture already includes a disabling mechanism, the proposed disabling scheme can be implemented by simply modifying the control module to also set the disable signal for the error flag, as shown in Figure 5.

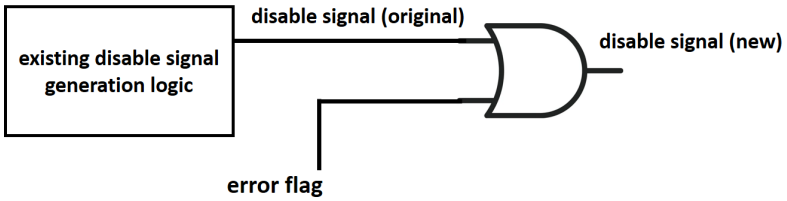

Figure 5. Extra hardware for generating the disable signal.

In the recurrent architecture, the error flag sets the skip signal. For the DT shown in Figure 6, if node 11 is the final leaf node, the DT hardware processes four nodes (node 1, 2, 4, and 8) sequentially. Meanwhile, if the potential error flag is set while processing node 2, processing node 4 and 8 is skipped, and processing the next input vector can start immediately, saving energy and reducing classification latency simultaneously. As the control module in the recurrent architecture already includes a stopping mechanism, the proposed stopping scheme can be easily incorporated by modifying the control module to set the skip signal not only when a leaf node is reached but also when the error flag is set, as shown in Figure 7.

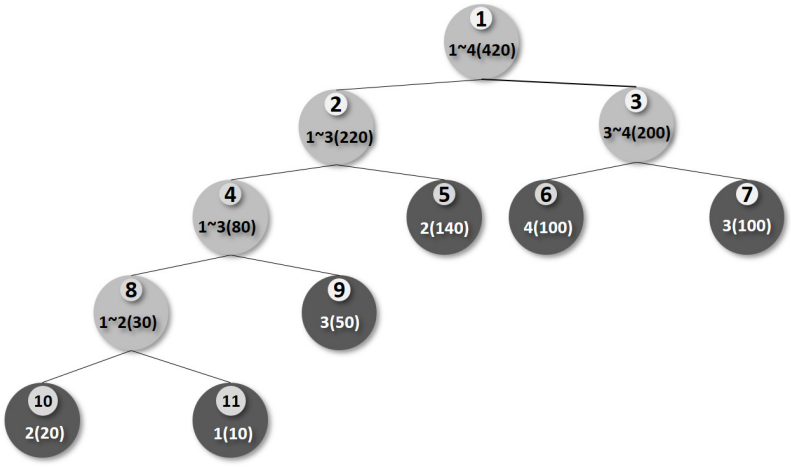

Figure 6. A DT example: the number inside each inner circle is a node number, the numbers before each left parenthesis indicate the class numbers of vectors visiting the corresponding node, the number inside each parenthesis is the number of input vectors visiting the corresponding node, and the darker circles are leaf nodes.

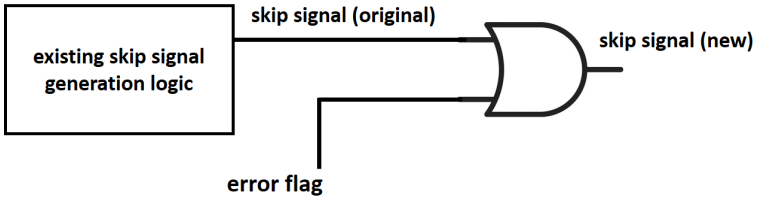

Figure 7. Extra hardware for generating the skip signal.

4.3. Classifying Detected Input Vectors

Even if a potential error is detected and the subsequent processing of the corresponding input vector is disabled or skipped, the class of the input vector still needs to be provided. For this, a probability-based class prediction method is proposed. The motivation for this method is based on the observation that it is possible to predict the class of an input vector even before reaching a leaf node. This predictive capability stems from the nature of the DT as a tree-based algorithm. As the traversal progresses down the tree, the potential terminal nodes where the traversal might conclude are progressively narrowed down. This narrowing of possibilities enhances the likelihood of correctly predicting the class before reaching a leaf node. This characteristic distinguishes DT from other machine learning algorithms like neural networks and support vector machines. By leveraging this unique feature of DT, our probability-based class prediction method offers a way to anticipate the likely class of an input vector, facilitating accurate classifications even for vectors that are flagged as potentially misclassified and have their processing interrupted.

Let us illustrate how class predictability is improved as tree traversal progresses. In Figure 6, at node 1, the probability that an input vector is classified as class 1 is 2.4% (10/420), as there are 10 train vectors of class 1 out of the total 420 train vectors that visited node 1. Similarly, the probability for an input vector to be classified as class 2, class 3, and class 4 is 38.1% (160/420), 35.7% (150/420), and 23.8% (100/420), respectively. Conversely, at node 4, the probability that an input vector is classified as class 1, class 2, and class 3 is 12.5% (10/80), 25.0% (20/80), and 62.5% (50/80), respectively. Therefore, if we predict the class of an input vector at node 1, we would predict it to be class 2 with a 38.1% likelihood. However, if we predict the class of an input vector at node 4, we would predict it to be class 3 with a 62.5% likelihood. It is important to note that these computations are based on the tree traversal history recorded using the training data.

This probability-based class prediction algorithm is as follows. First, we count the number of train vectors visiting each non-leaf node and record their class numbers during the training phase to compute the probabilities of each class at every non-leaf node. After computing class probabilities at every non-leaf node, the class with the highest probability (i.e., class 3 for node 4) is stored in its corresponding node memory, and this is used as the classification output for the input vectors that make the error flag be set.

As shown in Figure 8, this mechanism can be integrated into the baseline DT architectures by adding a register for a probabilistically determined class number, another register for an invalid class number, and a multiplexer that chooses between the class number of the corresponding leaf node and the probabilistically chosen class number. The select signal to the multiplexer is set to one, and the probabilistically selected class is chosen only if the selected class number is valid and the error flag is set. Note that the equality comparator produces one if the preselected class is equal to the invalid class number, which is set to a value larger than all valid class numbers.

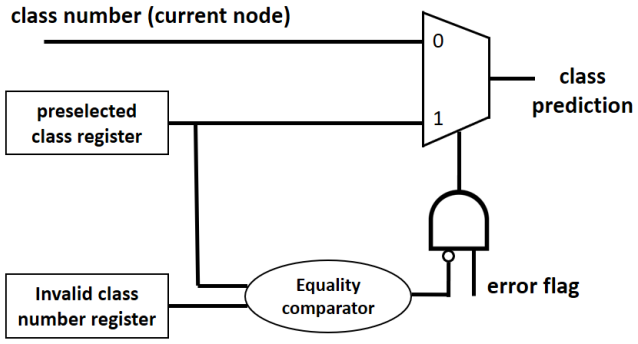

Figure 8. Extra hardware for probability-based class prediction.

5. Experiments

To evaluate the effectiveness of the proposed architectures, we employed public domain datasets from [32]. As a preprocessing step, feature-wise normalization was conducted to ensure that the data fell within the range of -1 to 1. Subsequently, each dataset was divided into four groups, with two groups being allocated for training and the remaining two for validation and testing, respectively. With six distinct combinations of two groups, six unique training datasets were created, each of which were employed in a cross-validation run (specifically, a six-fold cross-validation). In each cross-validation run, a tree model was learned and tested, yielding six sets of experimental results, such as accuracies and ratios. Consequently, most of the results presented in this paper represent averages derived from six different values.

DT training was performed using floating-point data utilizing the OC1 algorithm [33]. The hyperparameter controlling the proportion of the training set used in pruning was tuned using a grid search with the validation data. The search was to find the model with the highest accuracy. If there was a tie in terms of accuracy, the model with the highest diversion ratio was selected. It is important to note that the resulting node coefficients from the training process were converted to fixed-point values offline before being presented to the node memory. Similarly, input vectors were provided in fixed-point format to the input memory. For real number representation, allocating 11 bits to the integer part and 5 bits to the fractional part proved sufficient to achieve satisfactory classification accuracy for all datasets.

Every DT architecture was implemented using the Verilog hardware description language, and the implementation codes underwent simulation, synthesis, and implementation using Xilinx's Vivado Design Suite, targeting the Kintex-7 family FPGA. This comprehensive evaluation process ensures a thorough examination of the proposed architectures across a diverse set of datasets. Note that there are 168 DT architectures in total: 6 validation runs, 14 datasets, and 2 DT architectures. The oblique DT classifier code was written in C programming language and was used to generate all experimental results other than hardware-related ones.

Table 2 shows the tree height, number of nodes in each tree, number of attributes, and number of classes for each dataset. Note that the tree heights and node counts are presented as average values computed over six distinct tree models, each corresponding to a cross-validation run. These values are rounded off to the nearest integers. Upon inspection of the table, a significant variation is evident in both the number of nodes in a tree and the number of attributes across different datasets. Specifically, the number of nodes in a tree ranges from 6 (*Iris*) to 38 (*Car*), while the number of attributes varies from 3 (*Survival* and *Haber*) to 33 (*Ion*). This highlights the unique characteristics inherent in each dataset. Despite the diversity, a common observation is that taller trees tend to contain more nodes, reflecting a general trend across the datasets.

Table 2. Tree model size (height and node count) and basic dataset information.

Dataset	Tree Height	Node Count	Num. of Attributes	Num. of Classes
Car	7	37	6	4
Ecoli	6	28	7	8
Wine	3	7	13	3
Survival	5	15	3	2
Diabetes	6	26	8	2
Iris	3	6	4	3
Hayes	4	12	5	3
Scale	5	16	4	3
BC	4	12	9	3
Ion	5	14	33	2
Glass	6	26	9	6
Haber	6	19	3	2
Liver	7	28	6	2
Raisin	6	21	7	2
Average	6	24	8.2	3.2

5.1. Effectiveness of Potential Error Detection

The effectiveness of the potential error detection mechanism is crucial in the proposed architectures because it affects the usefulness of the disabling and skipping schemes. The mechanism should be able to detect input vectors that are likely to experience different node traversal. Table 3 shows how many input vectors are detected as potentially erroneous (*Det. ratio*), how many input vectors with diversions are detected (*Det. coverage (diversion)*), how many input vectors with a class switch are detected (*Det. coverage (class)*), and how many false alarms exist in the potential error detection (*False alarm ratio*). Note here that an input vector is considered as potentially erroneous if a potential error is detected while processing the input vector. The last column of the table indicates how much reduction is achieved in terms of the number of visited nodes if we stop processing an input vector once it is detected as potentially erroneous. Because no node diversion and class switch occur for *BC*, both detection coverage ratios of *BC* are *N/A*.

While majority of node diversions and class switches are detected successfully, the detection ratios seem to be high compared with the diversion ratios presented in Table 1. This observation aligns with the high false alarm ratios reported in Table 3. However, it should be noted that a high false alarm ratio is not necessarily detrimental because it provides more opportunities for subsequent disabling and skipping schemes. It is important to highlight that the detection mechanism does not flag arbitrary input vectors as potentially erroneous. Instead, it identifies input vectors that are inherently more challenging to be correctly classified. This nuanced approach to detection, targeting vectors with higher classification difficulty, will be further elucidated in the subsequent section.

The reduction ratios generally exhibit a proportional relationship to the detection ratios, but there are a few exceptions, such as the cases with *Hayes* and *Scale*. Despite having similar reduction ratios, these datasets display different detection ratios. Specifically, the ratio of *Hayes* is approximately twice as high as that of *Scale*. This discrepancy is associated with the timing of the potential error detection. When a potential error is detected earlier in the decision tree traversal, it results in the skipping of more nodes. Conversely, if the detection occurs near a leaf node, fewer nodes will be skipped. Therefore, the variation in detection ratios for different datasets can be attributed to differences in how early potential errors are identified during the classification process.

Table 3. Effectiveness of the potential error detection mechanism.

Dataset	Det. Ratio	Det. Coverage (Diversion)	Det. Coverage (Class)	False Alarm Ratio	Reduction Ratio
Car	0.151	0.808	0.833	0.940	0.096
Ecoli	0.310	0.877	0.882	0.648	0.165
Wine	0.163	1.000	1.000	0.791	0.093
Survival	0.367	0.756	1.000	0.779	0.252
Diabetes	0.398	0.949	0.880	0.643	0.213
Iris	0.112	0.667	0.667	0.839	0.058
Hayes	0.263	0.708	0.625	0.848	0.15
Scale	0.137	0.900	1.000	0.920	0.142
BC	0.153	N/A	N/A	1.000	0.090
Ion	0.518	0.911	0.850	0.919	0.310
Glass	0.277	0.817	0.792	0.708	0.127
Haber	0.255	0.711	0.778	0.682	0.163
Liver	0.467	0.910	0.917	0.627	0.258
Raisin	0.219	0.806	0.781	0.484	0.216
Average	0.265	0.833	0.849	0.756	0.162

5.2. Effectiveness of Probability-Based Class Prediction

Indeed, the proposed probability-based class prediction and the subsequent disabling/skipping mechanisms serve the crucial function of optimizing the utilization of opportunities created by the potential error detection mechanism. By accurately predicting the class of an input vector flagged as potentially misclassified, the probability-based class prediction method ensures that even if the processing of the vector is halted or skipped, a meaningful and informed class assignment can be provided. A reduction in energy and execution time is directly attributable to the disabling and skipping mechanism, but without the probability-based class prediction, some of the classification outputs cannot be provided, impairing the practicality of the DT classifier.

Table 4 shows how the probability-based class prediction mechanism performs compared with the baseline fixed-point DT. The *Prediction accuracy* column contains the prediction accuracies of the probability-based class prediction for the input vectors that are detected by the potential error detection mechanism. The classification accuracies of the baseline fixed-point DT for the same input vectors are presented in *Original accuracy*. The *Overall accuracy* column holds the classification accuracies of the baseline fixed-point DT for all input vectors. This column is added to the table to be compared with the *Original accuracy* column so that the effectiveness of the potential error detection mechanism can be revealed. The ratios presented in the *Both correct* column indicate how many of the input vectors are correctly classified by both the original classification method and the proposed probability-based class prediction. Similarly, the *Both incorrect* column contains ratios for input vectors that are incorrectly classified by both classification methods. The *I2C* column depicts ratios for input vectors that are incorrectly classified by the baseline fixed-point DT but are correctly classified by the probability-based class prediction. Conversely, the *C2I* column represents ratios of input vectors that are correctly classified by the baseline fixed-point DT but are incorrectly classified by the probability-based class prediction. These detailed metrics provide insights into the specific contributions and areas of improvement offered by the proposed mechanisms, shedding light on how effectively they address misclassifications identified through the potential error detection mechanism.

The table reveals that the probability-based classification outperforms the baseline fixed-point DT for input vectors flagged as potentially erroneous. This observation holds true across all datasets except for *Car*, *Iris*, *BC*, and *Ion*. On average, the probability-based classification exhibits a 7.7% higher accuracy compared with the baseline fixed-point DT for the identified flagged input vectors. It is important to note that because both classification mechanisms produce identical results for input vectors not detected by the potential error detection mechanism, the overall accuracy of the proposed architectures surpasses that of the baseline fixed-point DT architectures. This is corroborated by the results presented in Table 5, underscoring the effectiveness of the proposed mechanisms in improving the overall accuracy of the DT implementations.

Table 4. Effectiveness of probability-based class prediction (I2C: incorrect to correct; C2I: correct to incorrect).

Dataset	Prediction Accuracy	Original Accuracy	Overall Accuracy	Both Correct	Both Incorrect	I2C	C2I
Car	0.744	0.778	0.915	0.678	0.156	0.066	0.100
Ecoli	0.642	0.606	0.764	0.468	0.220	0.174	0.138
Wine	0.857	0.735	0.928	0.641	0.049	0.216	0.094
Survival	0.789	0.742	0.724	0.710	0.180	0.078	0.031
Diabetes	0.640	0.606	0.697	0.392	0.146	0.248	0.215
Iris	0.781	0.831	0.951	0.781	0.169	0.000	0.050
Hayes	0.547	0.424	0.672	0.331	0.360	0.215	0.093
Scale	0.690	0.668	0.913	0.551	0.193	0.139	0.117
BC	0.864	0.889	0.937	0.843	0.090	0.021	0.046
Ion	0.883	0.883	0.884	0.867	0.101	0.017	0.016
Glass	0.625	0.581	0.647	0.535	0.329	0.090	0.046
Haber	0.810	0.586	0.699	0.474	0.079	0.336	0.112
Liver	0.658	0.547	0.632	0.351	0.146	0.307	0.196
Raisin	0.715	0.644	0.845	0.581	0.222	0.134	0.063
Average	0.724	0.672	0.794	0.573	0.177	0.151	0.099

The noticeable accuracy discrepancy (18%) between the *Original accuracy* and *Overall accuracy* columns implies that the potential error detection mechanism does not flag any arbitrary input vectors but flags input vectors that are harder to predict correctly (in other words, easier to be incorrectly classified). Because the classification accuracy of the baseline fixed-point DT for flagged input vectors (*Original accuracy*) is relatively low, the probability-based class prediction mechanism does not need to be state of the art to match the accuracy of the baseline fixed-point DT. This is an important advantage for the potential error detection mechanism; a high false alarm ratio without this characteristic may make the probability-based class prediction fail to catch up with the classification accuracy of the baseline DT. A crucial observation from the table is that if the proportion of *I2C* is larger than that of *C2I*, it signifies that the number of input vectors whose classification outcome is changed from being incorrect to being correct by the probability-based class prediction outweighs the number of input vectors whose classification outcome is changed from being correct to being incorrect by the probability-based class prediction. This dynamic results in an overall improvement in classification accuracy, further highlighting the positive impact of the probability-based class prediction on the DT implementation.

5.3. Effectiveness of the Proposed Architectures

The efficacy of the proposed DT architectures is summarized in Table 5, encompassing classification accuracy, dynamic energy consumption, resource utilization, and execution time. Dynamic energy was estimated by the power estimation tool of the Vivado Design Suite, utilizing simulation activity files from functional simulations. Execution time was estimated by multiplying the simulation clock cycle counts and clock periods obtained from the Vivado timing analysis. Each result in the table is an average value computed from six validation runs.

Table 5. Effectiveness of the proposed architectures: accuracy, energy estimation, and execution time.

Dataset	Accuracy		Dynamic Energy		Resource (LUT/Register)		Exe. Time
	Fixed-Point	Proposed	Disabling	Skipping	Disabling	Skipping	Skipping
Car	1.002	0.996	0.946	0.927	1.023/1.006	1.003/1.000	0.841
Ecoli	0.987	1.013	0.785	0.883	1.044/1.003	1.013/1.001	0.840
Wine	0.992	1.000	1.005	0.929	1.010/1.001	1.008/1.000	0.905
Survival	0.988	1.006	0.938	0.775	1.101/1.005	1.027/1.001	0.751
Diabetes	0.983	1.001	0.877	0.804	1.020/1.005	1.006/1.001	0.666
Iris	1.005	1.000	1.084	0.976	1.024/1.004	1.024/1.001	0.933
Hayes	1.008	1.045	0.813	0.880	1.030/1.006	1.021/1.002	0.792
Scale	1.001	1.004	1.000	0.877	1.036/1.007	1.010/1.001	0.838
BC	1.001	0.995	1.005	0.920	1.028/1.002	1.005/1.000	0.875
Ion	1.011	1.011	0.685	0.696	1.006/1.001	1.002/1.000	0.660
Glass	1.004	1.019	0.960	0.893	1.032/1.003	1.014/1.001	0.852
Haber	0.984	1.031	0.941	0.863	1.084/1.007	1.024/1.002	0.827
Liver	0.931	1.017	0.827	0.758	1.061/1.004	1.012/1.001	0.706
Raisin	0.987	1.000	0.937	0.792	1.049/1.004	1.006/1.001	0.805
Average	0.992	1.010	0.915	0.855	1.039/1.004	1.012/1.001	0.807

The two columns of *Accuracy* show the classification accuracies of the baseline fixed-point oblique DTs [25] and the proposed DTs. The normalization of values in these columns to the accuracies of software floating-point oblique decision trees (DTs) explains why some accuracies may be greater than 1. In this context, an accuracy greater than 1 signifies that the corresponding fixed-point DT outperforms the floating-point DT in terms of accuracy. Normalization to a reference value—in this case, the accuracy of the software floating-point DT—allows for a comparative analysis, facilitating the assessment of how well the fixed-point DTs perform relative to the floating-point counterpart. This approach provides a standardized metric for evaluating the accuracy improvement achieved by the proposed fixed-point DT architectures.

As shown in the columns, both fixed-point implementations closely match the floating-point counterpart. For certain datasets (*Hayes*, *Scale*, *Ion*, and *Glass*), both fixed-point DTs exhibit higher accuracies than the floating-point DT. This anomaly can be attributed to the fact that switched classification outputs due to rounding inadvertently convert incorrect classifications to correct classifications more frequently than they convert correct classifications to incorrect classifications. Regarding the proposed DT, the probability-based class prediction mechanism plays a role too. On average, the accuracies of the proposed DTs are slightly higher than those of the baseline fixed-point DTs. This observation affirms the effectiveness of the potential error detection and the probability-based class prediction mechanisms in improving the overall accuracy of the DT implementations.

The dynamic energy consumption of the proposed DT is presented in the columns denoted as *Disabling* and *Skipping*. Note that *Disabling* and *Skipping* denote the proposed pipeline and recurrent DT architectures, respectively. The values in these columns are the energy consumption ratios, which were normalized to the energy consumptions of the baseline fixed-point pipeline architecture and recurrent architecture, respectively. As seen from the table, both schemes reduce energy consumption, and the amount of the reduction is proportional to the traversed node reduction ratio (*Reduction ratio*) in Table 3. In fact, the reduction in energy consumption is less than the traversed node reduction ratio because of the extra hardware added to implement the proposed mechanisms. Because of the extra hardware, the proposed schemes consume a bit more energy than the baseline fixed-point DTs for *Wine*, *Iris*, and *BC*, for which the amount of disabled or skipped computations is relatively small.

In the *Resource(LUT/Register)* column, the number of utilized slice LUTs and slice registers are presented. The values in the column are normalized to the number of LUTs and registers utilized to implement the baseline fixed-point DT architectures. From the table, it is confirmed that the extra hardware of the proposed DT architectures can be implemented without consuming too much of the FPGA resource.

Execution time reduction is observed only from the recurrent architecture because in the pipeline architecture, every input vector has to go through all pipeline stages, even when the potential error flag is set. The amount of execution time reduction is also proportional to the traversed node reduction ratio shown in Table 3.

In summary, the proposed fixed-point oblique DT architectures demonstrate classification accuracies comparable with their software implementation counterparts. While they incur a slightly higher FPGA resource requirement than the baseline DT architectures due to the incorporation of additional hardware mechanisms, these extra features enable the proposed DTs to achieve classification speeds that are 21.6% faster while consuming less dynamic energy. The dynamic energy reduction is measured at 8.3% and 14.2% for the pipeline and recurrent architectures, respectively.

6. Conclusions

The decision tree is a simple yet versatile classification algorithm. Its simplicity and commendable classification performance make it a frequent choice for hardware implementation in embedded systems. In this context, we present efficient implementations of a fixed-point oblique DT. The proposed architectures feature hardware enhancements designed by leveraging the algorithmic features of DTs, specifically focusing on the computation at each node for child node determination and the tree traversal from the root node to a leaf node. Firstly, utilizing the node computation feature, we detect an input vector that may lead to a rounding-induced tree traversal diversion and flag it as erroneous. This detection mechanism successfully identifies approximately 85% of node diversions. Subsequently, further processing of the vector is either disabled or skipped depending on the architecture, thereby enhancing classification speed and energy efficiency. Lastly, to provide a classification output for a flagged and interrupted input vector, we employ a probability-based class prediction method. This method maintains the overall classification accuracy at the same level as existing fixed-point oblique DT architectures. This enhancement is rooted in a DT algorithm's inherent characteristic: DT classification is conducted step by step by traversing tree nodes. In comparison with existing fixed-point implementations, the proposed implementation demonstrates a 23.9% faster classification speed (specifically for the recurrent architecture) while consuming 11.5% less dynamic energy, all without sacrificing classification accuracy. These enhancements have the potential to improve user experience, leading to more responsive services and extended battery life, particularly if the proposed designs are integrated into smart embedded devices. Although the proposed architectures are designed for a single DT, they are expected to be versatile enough for use in ensemble classifiers as a fundamental classifier design. Furthermore, they can be seamlessly integrated with other hardware-centric implementation techniques.

The future enhancement of the proposed architectures will be explored along three main directions. Firstly, the probability-based class prediction can undergo refinement to achieve higher accuracy. Because this prediction is carried out offline during the training phase, a more sophisticated prediction algorithm can be employed without impacting the DT hardware. Secondly, further improvement is conceivable through potential error detection. As illustrated in Section 4.1, f^{up} is approximated by adding n×SPV to f^{down}, where n represents the number of multiplications in node computation. In the original mechanism, n is set to the number of attributes for each dataset. However, optimizing n for each dataset is a potential enhancement, offering the possibility of improving the detection ratio and/or classification accuracy. This optimization allows for a more tailored approach to potential error detection, adapting to the characteristics of individual datasets and thereby potentially enhancing the overall performance of the system. Lastly, an exploration will be conducted on an implementation utilizing dual fixed-point representation [34]. This approach combines the simplicity of fixed-point arithmetic with the broader dynamic range of floating-point arithmetic. The investigation will assess accuracy, classification speed, and energy consumption, with the aim of identifying a more suitable DT implementation for embedded systems.

Funding: This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Education (NRF-2014R1A1A2058198).

Data Availability Statement: UCI repository of machine learning databases: http://archive.ics.uci.edu/ml (accessed on 20 August 2022).

Conflicts of Interest: The author declares no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

DT	Decision tree
SoC	System-on-chip
FPGA	Field-programmable gate array
SPV	Smallest positive value
LUT	Lookup table

References

1. Zhao, H.; Liu, J.; Chen, H.; Chen, J.; Li, Y.; Xu, J.; Deng, W. Intelligent diagnosis using continuous wavelet transform and gauss convolutional deep belief network. *IEEE Trans. Reliab.* **2023**, *72*, 692–702. [CrossRef]
2. Li, M.; Zhang, W.; Hu, B.; Kang, J.; Wang, Y.; Lu, S. Automatic assessment of depression and anxiety through encoding pupil-wave from HCI in VR scenes. *ACM Trans. Multimed. Comput. Commun. Appl.* **2023**, *20*, 1–22. [CrossRef]
3. Chiang, Y.H.; Lin, Y.R.; Chen, N.S. Using deep learning models to predict student performance in introductory computer programming courses. In Proceedings of the 2022 International Conference on Advanced Learning Technologies, Bucharest, Romania, 1–4 July 2022; pp. 180–182.
4. Samia, B.; Soraya, Z.; Malika, M. Fashion images classification using machine learning, deep learning and transfer learning models. In Proceedings of the 2022 International Conference on Image and Signal Processing and their Applications, Mostaganem, Algeria, 8–9 May 2022; pp. 1–5.
5. Breiman, L. *Classification and Regression Trees*, 1st ed.; Taylor & Francis Group: New York, NY, USA, 1984; pp. 18–58.
6. Wang, S.; Fan, C.; Hsu, C.H.; Sun, Q.; Yang, F. A vertical handoff method via self-selection decision tree for internet of vehicles. *IEEE Syst. J.* **2016**, *10*, 1183–1192. [CrossRef]
7. Duanmu, F.; Ma, Z.; Wang, Y. Fast mode and partition decision using machine learning for intra-frame coding in HEVC screen content coding extension. *IEEE J. Emerg. Sel. Top. Circuits Syst.* **2016**, *6*, 517–531. [CrossRef]
8. McCarthy, M.W.; James, J.A.; Lee, J.B.; Rowlands, D.D. Decision-tree-based human activity classification algorithm using single-channel foot-mounted gyroscope. *Electron. Lett.* **2015**, *51*, 675–676. [CrossRef]
9. Piao, M.; Jin, C.H.; Lee, J.Y.; Byun, J.Y. Decision tree ensemble-based wafer map failure pattern recognition based on radon transform-based features. *IEEE Trans. Semicond. Manuf.* **2018**, *31*, 250–257. [CrossRef]
10. Rahmatillah, I.; Astuty, E.; Sudirman, I.D. An Improved Decision Tree Model for Forecasting Consumer Decision in an Medium Groceries Store. In Proceedings of the 2023 International Conference on Industrial and Information Systems, Peradeniya, Sri Lanka, 25–26 August 2023; pp. 245–250.

11. Albia, C.J.D.; Origines, D. Selecting of pet adopters using C4.5 decision tree model algorithm. In Proceedings of the 2022 International Conference in Information and Computing Research, Cebu, Philippines, 10–11 December 2022; pp. 30–35.
12. Winkler, J.; Lunglmayr, M. FPGA processing of decision tree ensembles stored in external DRAM. In Proceedings of the 2023 International Conference on Electrical, Computer, Communications and Mechatronics Engineering, Tenerife, Canary Islands, Spain, 19–21 July 2023; pp. 1–6.
13. Bartels, J.; Tokgoz, K.K.; Fukawa, M.; Otsubo, S.; Chao, L.; Rachi, I.; Takeda, K.; Ito, H. A 216 uW, 87% accurate cow behavior classifying decision tree on FPGA with interpolated arctan2. In Proceedings of the 2021 IEEE International Symposium on Circuits and Systems, Daegu, Republic of Korea, 22–28 May 2021; pp. 1–5.
14. Alcolea, A.; Resano, J. FPGA accelerator for gradient boosting decision trees. *Electronics* **2021**, *10*, 314. [CrossRef]
15. Shoaran, M.; Haghi, B.A.; Taghavi, M.; Farivar, M.; Emami-Neyestanak, A. Energy-efficient classification for resource-constrained biomedical applications. *IEEE J. Emerg. Sel. Top. Circuits Syst.* **2018**, *8*, 693–707. [CrossRef]
16. Tong, D.; Sun, L.; Matam, K.; Prasanna, V. High throughput and programmable online traffic classifier on FPGA. In Proceedings of the 2013 ACM/SIGDA International Syposium on FPGA, Monterey, CA, USA, 11–13 February 2013; pp. 255–264.
17. Oberg, J.; Eguro, K.; Bittner, R.; Forin, A. Random decision tree body part recognition using FPGAs. In Proceedings of the 2012 International Conference on Field Programmable Logic and Applications, Oslo, Norway, 29–31 August 2012; pp. 330–337.
18. Li, Q.; Bermak, A. A low-power hardware-friendly binary decision tree classifier for gas identification. *J. Low Power Electron. Appl.* **2011**, *1*, 45–58. [CrossRef]
19. Murthy, S.K.; Kasif, S.; Salzberg, S. A system for induction of oblique decision trees. *J. Artif. Intell. Res.* **1994**, *2*, 1–32. [CrossRef]
20. Zhang, L.; Suganthan, P.N. Oblique decision tree ensemble via multisurface proximal support vector machine. *IEEE Trans. Cybern.* **2015**, *45*, 2165–2176. [CrossRef] [PubMed]
21. Tang, Y.; Verma, N. Energy-efficient pedestrian detection system: Exploiting statistical error compensation for lossy memory data compression. *IEEE Trans. Vlsi Syst.* **2018**, *26*, 1301–1311. [CrossRef]
22. Nakahara, H.; Jinguji, A.; Sato, S.; Sasao, T. A random forest using a multi-valued decision diagram on an FPGA. In Proceedings of the IEEE International Symposium on Multiple-Valued Logic, Novi Sad, Serbia, 22–24 May 2017; pp. 266–271.
23. Owaida, M.; Zhang, H.; Zhang, C.; Alonso, G. Scalable inference of decision tree ensembles: Flxible design for CPU-FPGA platforms. In Proceedings of the International Conference on Field Programmable Logic and Applications, Ghent, Belgium, 4–8 September 2017; pp. 1–8.
24. Qu, Y.R.; Prasanna, V.K. Scalable and dynamically updatable lookup engine for decision-trees on FPGA. In Proceedings of the IEEE High Performance Extreme Computing Conference, Waltham, MA, USA, 9–11 September 2014; pp. 1–6.
25. Struharik, J.R. Implementing decision trees in hardware. In Proceedings of the IEEE International Symposium on Intelligent Systems and Informatics, Subotica, Serbia, 8–10 September 2011; pp. 41–46.
26. Zeng, Z.; Sapatnekar, S.S. Energy-efficient hardware acceleration of shallow machine learning applications. In Proceedings of the 2023 Design, Automation & Test in Europe Conference & Exhibition, Antwerp, Belgium, 17–19 April 2023; pp. 1–6.
27. Summers, S.; Guglielmo, G.D.; Duarte, J.; Harris, P.; Hoang, D.; Jindariani, S.; Kreinar, E.; Loncar, V.; Ngadiuba, J.; Pierini, M.; et al. Fast inference of boosted decision trees in FPGAs for particle physics. *J. Instrum.* **2020**, *15*, 1–11. [CrossRef]
28. Buschjager, S.; Morik, K. Decision tree and random forest implementations for fast filtering of sensor data. *IEEE Trans. Circuits Syst.* **2018**, *65*, 209–222. [CrossRef]
29. Struharik, R. Decision tree ensemble hardware accelerators for embedded applications. In Proceedings of the IEEE International Symposium on Intelligent Systems and Informatics, Subotica, Serbia, 17–19 September 2015; pp. 101–106.
30. Molina, R.; Loor, F.; Gil-Costa, V.; Nardini, F.M.; Perego, R.; Trani, S. Efficient traversal of decision tree ensembles with FPGAs. *J. Parallel Distrib. Comput.* **2021**, *155*, 38–49. [CrossRef]
31. Lucchese, C.; Nardini, F.M.; Orlando, S.; Perego, R.; Tonellotto, N.; Venturini, R. QuickScorer: A fast algorithm to rank documents with additive ensembles of regression trees. In Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval, Santiago, Chile, 9–13 August 2015; pp. 73–82.
32. UCI Repository of Machine Learning Databases. Available online: http://archive.ics.uci.edu/ml (accessed on 20 August 2022).
33. Murthy, S.; Kasif, S.; Salzberg, S.; Beigel, R. OC1: Randomized induction of oblique decision trees. In Proceedings of the National Conference on Artificial Intelligence, Washington, DC, USA, 11–15 July 1993; pp. 322–327.
34. Ewe, C.T. Dual fixed-point: An efficient alternative to floating-point computation for DSP applications. In Proceedings of the International Conference on Field Programmable Logic and Applications, Tampere, Finland, 24–26 August 2005; pp. 715–716.