TinyML: Applications, Algorithms, Co-design and Implementations

Omar Ghazal, Tousif Rahman and Rishad Shafik

Big things have small beginnings
Lawrence of Arabia (1962)

All Authors: Newcastle University, Newcastle upon Tyne (GB), e-mail: s.rahman@newcsatle.ac.uk

Fig. 1: A DALL-E 3 generated image for TinyML when given the prompt: *"Generate a picture of TinyML suitable as the introduction page of a book chapter."* It also produces the following accompanying caption: "This image visually symbolizes the integration of machine learning into small, resource-constrained devices like microcontrollers and smart sensors".

Abstract *Tiny Machine Learning* (*TinyML*) is a subset of Machine Learning (ML) application deployment where design focus is placed on transitioning memory and compute intensive ML models, typically trained on the cloud in large data-centres, into resource constrained *edge* devices. The motivation for this transition is to enable better energy efficiency, preserve user data security and minimize network usage. This chapter explores the deployment of a TinyML application through a pipeline of algorithm exploration, co-design techniques to optimize these algorithms for edge devices, and custom and commercial implementations of edge device architectures. In particular the chapter focuses on current developments with Neural Network

based approaches that offer state of the art performance and functionality as well as an emerging algorithm called the *Tsetlin Machine* to understand whether this recently proposed method ca be a suitable alternative. This algorithm-to-hardware approach is now developed into popular automation flows, libraries and toolchains; this chapter examines the scope, usability and drawbacks of these tools and platforms.

Contents

1 The TinyML Ecosystem

The rise of Internet-of-Things (IoT) based sensors and wearable devices has dramatically increased the ability to collect data at any time. Machine Learning (ML) provides a generalized processing method for this data to develop classification or prediction applications. This has had a profound impact in areas such as personalized healthcare, smart home monitoring, and predictive maintenance.

However, one of the caveats to the successful development of the aforementioned applications is translating the ML components into the IoT sensors and wearable devices themselves. The advantages of doing this are significant: The ML application can process potentially user-sensitive data in situ instead of transferring to a more powerful computing device through the network. This in situ approach is typically faster and more energy efficient and is known as ML on the *edge* (on edge devices). It can alleviate both network security concerns and bandwidth contention. The disadvantage of keeping the application within the sensor or the wearable itself comes from the resource limitations of these devices. This means that the ML model must fit into the edge device memory, the computation time must meet the application's latency requirement, and the model must provide adequate performance in its classification/prediction task. The trade-off between balancing the effectiveness of the ML application and whether it can be incorporated into the edge sensor or wearable is the principle crux of *Tiny Machine Learning* (*TinyML*). TinyML aims to bring intelligence to IoT devices, wearables, and other edge endpoints, enabling them to perform tasks autonomously without relying on cloud connectivity [1, 2].

The significance of TinyML in modern technology cannot be overstated. It offers a platform where machine learning models can be pushed to user devices, enhancing user experience across diverse applications while addressing concerns such as energy efficiency, reduced costs, data security, and low latency prevalent in contemporary cloud computing technology [3]. By enabling the deployment of machine learning models on tiny devices, TinyML contributes to widening access to applied machine learning, fostering the development of real-world applications and ethical considerations in technology through hands-on programming and deployment experiences [4]. Furthermore, TinyML plays a crucial role in the development of energy-efficient IoT sensors for computer vision applications, demonstrating its potential to improve the sustainability and effectiveness of IoT devices [5].

To further unpack these ideas, this chapter examines fundamental concepts laced with the latest developments in TinyML by presenting a pipeline that one might use when deploying a TinyML application. This is seen in Fig. 2. The left corner shows the main components of this deployment pipeline: Algorithm, Co-Design and Implementation, we briefly define them now:

Algortihms: This section briefly describes the compute, and architecture of the core and emerging algorithms that are used for TinyML. This includes the traditional ML methods like decision trees, random forests and linear regression, followed by deep learning approaches and the recently proposed Tsetlin Machine [6].

Co-Design: The goal of co-design is to translate the aforementioned algorithms into more memory and compute efficient implementations that can be accommodated

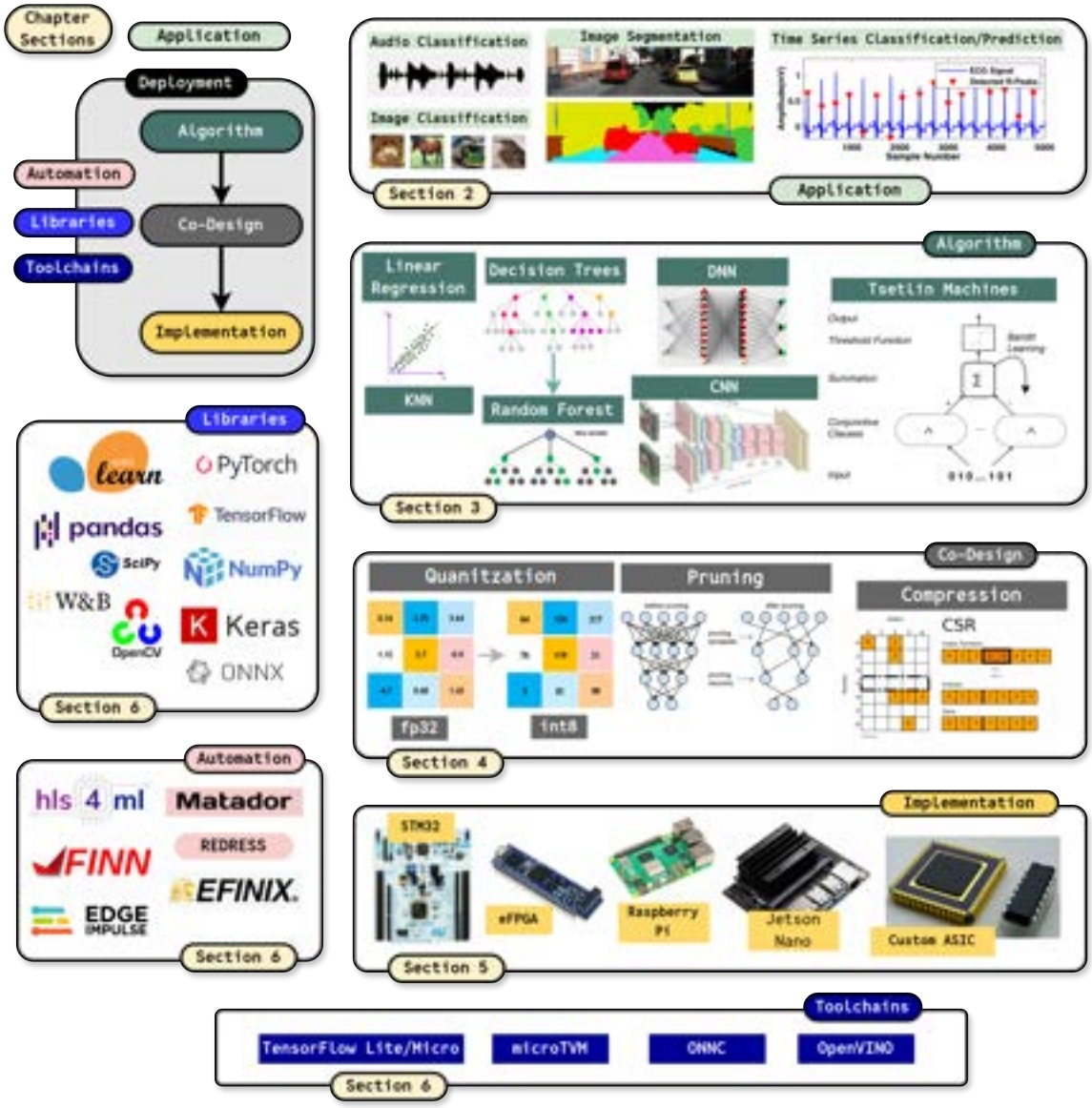


Fig. 2: A visual overview of the TinyML ecosystem of algorithms, co-design methods, implementations and the development tools and automation flows that link them together. This image also forms the organization of the sections presented in this chapter.

into the resource constrained edge nodes that are typically ubiquitous with IoT sensor systems, this includes small RISC processors with very limited flash and RAM packaged as development boards like the STM32 or ESP32 series.

Implementation: This section focuses on architectures that are used to implement the condensed models that were designed through the co-design methods. These architectures include CPU, GPU, TPU and FPGA based accelerators.

Automation, Libraries and Toolchains: Finally, the full flow of transition from algorithm to implementation is usually wrapped into an automation flow. This section explores the main components of these automation flows. By utilizing these lightweight frameworks, developers can efficiently deploy machine learning models on microcontrollers, allowing edge devices to perform complex AI tasks with minimal computational overhead [7]. The seamless integration of hardware and software components within the TinyML ecosystem is essential for enabling intelligent edge computing solutions.

The synergy of the components above form the TinyML *Ecosystem*. This ecosystem promotes innovation and collaboration among researchers, developers, and industry stakeholders, advancing edge computing and AI technologies. By democratizing machine learning on ubiquitous microcontrollers, TinyML has expanded access to AI capabilities, fostering the development of intelligent edge devices across industries. Collaborative efforts within the TinyML ecosystem have led to the emergence of novel approaches and solutions, such as Tiny Deep Learning Architectures and Physics-Enhanced TinyML, catering to specific application domains with unique requirements [8, 9]. This collaborative spirit and innovation within the TinyML ecosystem drive advancements in edge AI and the evolution of intelligent IoT solutions.

Chapter Organization: Fig. 2 also contains the organization of this chapter. The following section will discuss some suitable applications for TinyML and the type of ML task that they require. Section 3 examines the properties of commonly used ML algorithms in the context of TinyML. Section 4 presents optimization methods for these algorithms to transition them to TinyML devices. Section 5 explores the actual implementation of the ML model and the application on the mentioned devices. The remaining section explores how the previous four sections can be tied together for an easier transition or end-to-end automation.

2 Applications

The TinyML ecosystem extends to various use cases in wearables, smart home devices, and industrial IoT applications, demonstrating the versatility and applicability of TinyML technology [10]. In wearables, TinyML enables the implementation of AI algorithms on compact devices, enabling them to perform tasks such as user recognition and activity monitoring efficiently. Smart home devices benefit from TinyML by incorporating intelligent features like anomaly detection and energy optimization, improving functionality and user experience. Moreover, in industrial IoT

Table 1: Applications of TinyML in Various Industrial Sectors.

Sector	Applications
Industrial	Predictive maintenance (PdM) [14], Enhancing reproducibility [11], Gas leakage detection [15], Energy efficiency [16], Object detection [17], Obstacle detection [18].
Other	Smart greenhouse [12], Smart farming [19], Environmental monitoring [20], Adaptive resolution inference [21].
Consumer	Smart home devices [22], Safe lock entry [23], Voice command recognition [24], Multisensor fusion framework [25], Gas leakage detection [15], Diabetes detection [26].
Health	Healthcare [13], LPWAN wearable devices [27], COVID-19 disease detection [28], Air pollution prediction [29], Implantable devices [30].
Finance	Fraud detection [31], Credit risk assessment [32], Algorithmic trading [33].
Retail	Retailer supplier switching [34], Inventory optimization [35].
Government	Big data management [10], CCTV object detection [36], People counting [37], Hydraulic infrastructure health [38].
Energy	Green AI [39], Demand response [40], Energy conservation [41].
Transport	Mobility vehicles fall detection [42], Crowd monitoring [5].

settings, TinyML is applied in smart factories and predictive maintenance systems, enabling real-time data analysis and decision making at the edge [11, 12]. TinyML finds applications in wearable and ambulatory systems, enabling embedded machine learning on microcontrollers for health and care applications [13].

Table 1 summarizes some of these applications of TinyML across different domains, providing an overview of its diverse implementations.

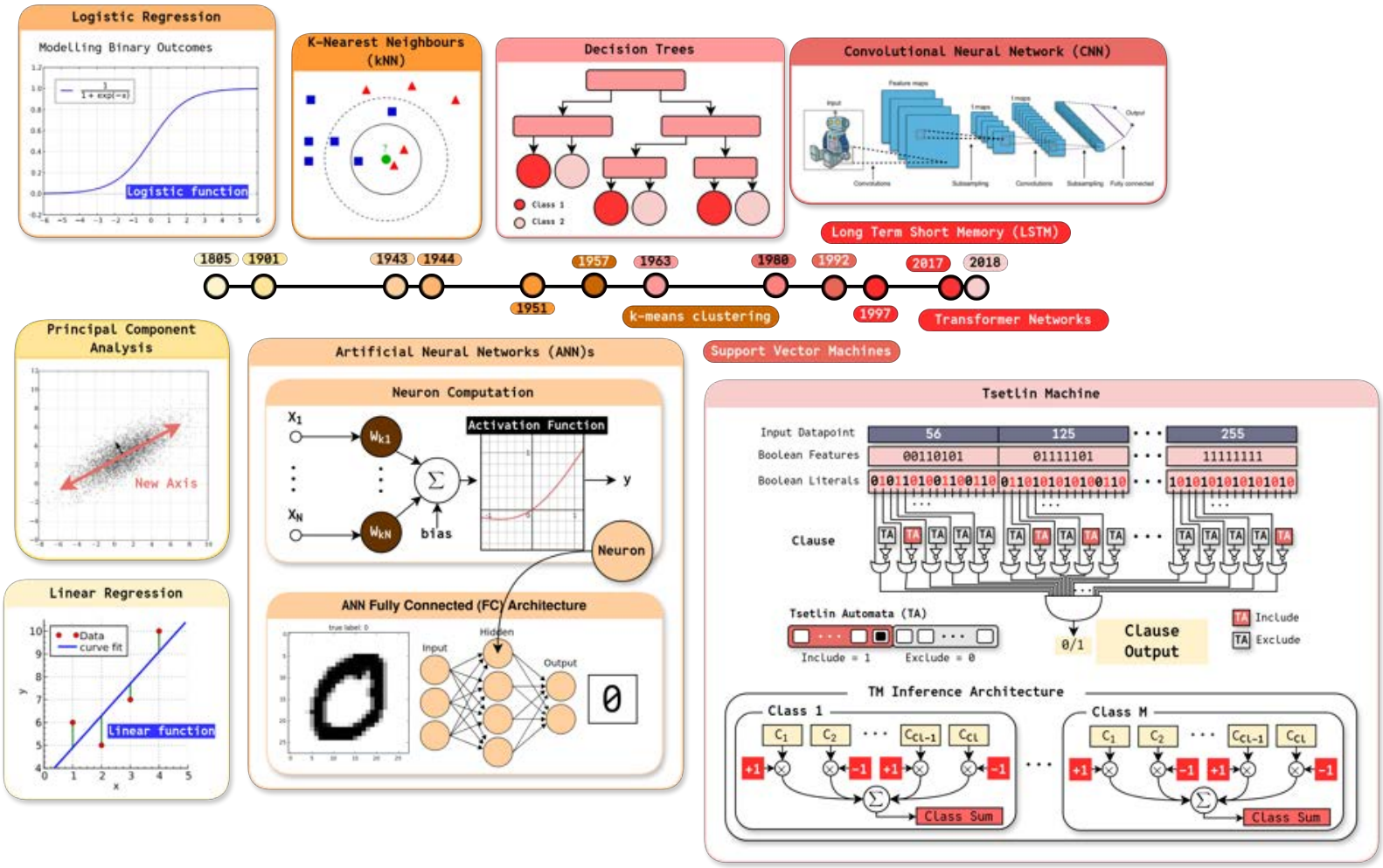


Fig. 3: Timeline showing the invention of core algorithms used in ML applications. The selected algorithms have been visually depicted in a stylized manner. Color coding is used to associate the algorithm with its place in the timeline.

3 Algorithms (Refresher)

Before discussing the theoretical concepts behind TinyML, it is important to quickly outline the fundamentals of widely used ML algorithms, this time from the perspective of TinyML. While reading this section, it is worth thinking about the following questions about each of the algorithms presented.

- How intensive is the computation - how much hardware would be required?
- How large is the model - how easily would it be accommodated into memory constrained implementations?
- How fast is the algorithm (this will be a combination of the first two points)?
- What are the TinyML performance metrics to determine whether a particular algorithm is most suited to be the classifier or regressor in the ML application?

Overview

This section provides a brief summary of traditional ML approaches, deep learning, and emerging methods in chronological order. Traditional ML uses statistical approaches to make predictions (e.g. linear regression, k-Nearest Neighbors). Deep learning allows for application development without as much design effort in feature engineering (e.g. Convolutional Neural Networks). For each algorithm, the model computation, size, speed, and metrics are given to determine their suitability for TinyML. Later sections will explore how the most suitable algorithms can be further developed into implementations.

This chapter presents a very high-level summary of these algorithms, where appropriate, more in-depth explanations have been referenced. The next three subsections will explore the algorithms presented in Fig. 3. The same chronology presented in the timeline in the figure will be used to explore how ML algorithms have transitioned from traditional statistical models to more intricate non-linear and compute heavy approaches to solve more complex problems. Finally, there will be an examination of one particular emerging algorithm called the *Tsetlin Machine (TM)*.

3.1 Traditional ML Algorithms

Machine Learning algorithms find their roots in statistical methods. Typically, datasets were constructed using domain-specific knowledge for feature extraction - this extraction process was crucial to the learning efficacy of traditional models [43]. This data would then be used in statistical analysis. With respect to Fig. 3, these are the early works of 1805-1901.

3.1.1 Linear Regression

One of the simplest algorithms, it models the relationship between a dependent variable and one or more independent variables by fitting a linear equation. The method of least squares, which is the foundation of linear regression, was first introduced by Adrien-Marie Legendre in 1805. Later, Carl Friedrich Gauss expanded on it in 1821, using the same principles for astronomical data fitting. Linear regression is highly suitable for TinyML. It requires minimal computational resources and can be easily deployed on microcontrollers. Its small memory footprint and fast inference times make it a great option for tasks such as sensor data analysis or predictive maintenance in IoT devices.

- **Computation:** Minimal computation is required, making it one of the most lightweight algorithms.
- **Model Size:** Extremely small, often requiring only coefficients that fit easily in memory-constrained environments.
- **Speed:** Very fast to train and predict, as it is based on simple matrix operations.
- **Metrics:** The most common metrics to evaluate regression tasks are the mean squared error (MSE), the root mean squared error (RMSE) and the R-squared. It is suitable when there is a linear relationship between the input features and the output.

3.1.2 Logistic Regression

Despite its name, logistic regression is a classification algorithm. It is used for binary classification problems. Logistic regression applies the logistic function to model the probability of a binary outcome, using the log-odds of the dependent variable as a linear function of the independent variables. Joseph Berkson coined the term ”logit” in 1944, but the logistic function itself was used earlier by Pierre-François Verhulst in 1838 to describe population growth. The logistic function was formalized for binary classification by David Cox in 1958 in his paper The Regression Analysis of Binary Sequences published in Journal of the Royal Statistical Society.

Like linear regression, logistic regression is lightweight and easy to implement on memory-constrained devices. Its binary classification capability makes it suitable for applications such as anomaly detection, where decisions need to be made quickly and with minimal processing power.

- **Computation:** Slightly more complex than linear regression, as it calculates probabilities, but still very lightweight.
- **Model Size:** Small, requiring storage for weights and bias terms, easily accommodated in memory-constrained environments.
- **Speed:** Fast, with performance comparable to linear regression.
- **Metrics:** Key metrics include accuracy, precision, recall, and F1-score. Logistic regression works well when there’s a linear decision boundary between classes.

These ideas are further discussed in "Machine Learning: A Probabilistic Perspective" (Murphy, 2022), which explores logistic regression in depth and compares it with other probabilistic models.

3.2 Principal Component Analysis

PCA is used for dimensionality reduction, identifying the axes of greatest variance in the data and projecting it onto fewer dimensions while retaining most of the original information. PCA was introduced by Karl Pearson as a technique for dimensionality reduction by finding the directions (principal components) that capture the most variance in data. Pearson's original work On Lines and Planes of Closest Fit to Systems of Points in Space was published in Philosophical Magazine in 1901.

PCA can be beneficial in TinyML when used for dimensionality reduction, especially for preprocessing large datasets on constrained devices. Reducing the number of features helps minimize the computational and memory overhead of downstream tasks.

- **Computation:** Relatively lightweight but can become computationally expensive for very high-dimensional data due to eigenvalue decomposition.
- **Model Size:** Small, as PCA reduces dimensionality, producing fewer components than the original features.
- **Speed:** Fast for low-dimensional data, but slows with increased dimensionality.
- **Metrics:**Explained variance ratio is used to determine how much variance is captured by each principal component. PCA is ideal for dimensionality reduction when feature correlation is high.

3.3 Decision Trees

Decision Trees are a tree-like model that recursively splits the data into subsets, making decisions based on the feature that provides the best split until it arrives at a final classification or prediction. The foundation for decision trees was laid by William Belson in 1963 when he developed a method for creating binary trees. Later, the Classification and Regression Trees (CART) algorithm was introduced by Breiman et al. in 1984.

Decision trees, especially shallow trees, are well-suited for TinyML. They can perform fast inference with low computational overhead. However, large or deep trees may require more memory and processing power, which might limit their use in very constrained environments.

- **Computation:** Training can be more computationally intensive, especially with deep trees, but remains manageable compared to deep learning models.

- **Model Size:** Decision trees can grow large depending on the depth and number of features, potentially becoming memory-intensive.
- **Speed:** Relatively fast for prediction once trained, though training time increases with data complexity.
- **Metrics:** Gini impurity, entropy, and accuracy are used for classification, while MSE and R-squared are common for regression. Decision trees are a good choice when interpretability is key.

3.4 k-Nearest Neighbors (kNN)s

kNN is a non-parametric algorithm used for classification and regression. For a given input, it looks at the k closest data points in the feature space and classifies the input based on the majority class among its neighbors or averages the values for regression. Evelyn Fix and Joseph Hodges introduced the kNN algorithm as a non-parametric classification method. The original paper Discriminatory Analysis: Non-parametric Discrimination: Consistency Properties was published in 1951 as part of the USAF School of Aviation Medicine Project RAND.

kNN is generally unsuitable for TinyML due to the memory-intensive nature of the algorithm, which requires storing the entire dataset and performing real-time distance calculations for each new prediction. However, optimizations such as reducing the number of stored data points or applying appropriate nearest-neighbor techniques could make it feasible for limited applications.

- **Computation:** Training is nearly instantaneous (just storing the dataset), but prediction is computationally intensive, as it involves distance calculations with the entire dataset.
- **Model Size:** Memory requirements are high, as the entire dataset must be stored.
- **Speed:** Slow for prediction due to the need to compute distances between the query and all data points.
- **Metrics:** Accuracy, precision, recall, and F1-score are common for classification. Euclidean distance is typically used to measure similarity.

3.4.1 K-Means Clustering

This unsupervised learning algorithm groups data points into k clusters by minimizing the distance between each data point and the centroid of its assigned cluster. The centroids are updated iteratively until the clusters converge. K-means can be adapted to TinyML for lightweight clustering tasks.

Since K-means only need to store cluster centroids rather than the entire dataset, it can be used for simple classification or clustering in resource-constrained environments. Its computational efficiency makes it a good candidate for on-device anomaly detection. The K-means clustering algorithm was first introduced by Stuart Lloyd in 1957 during his work at Bell Labs. However, it was not widely known until it was

rediscovered and published in 1982. Lloyd's 1957 work remained internal until it was republished as Least Squares Quantization in PCM in 1982 in IEEE Transactions on Information Theory.

- **Computation:** K-means is computationally efficient for small-to-medium datasets but can become slow with a large number of clusters or features.
- **Model Size:** Very small, as the model only stores the cluster centroids.
- **Speed:** Training can be slow when recalculating centroids iteratively, but the overall computational cost is moderate.
- **Metrics:** Inertia (sum of squared distances to cluster centers) and silhouette score are common metrics to evaluate clustering performance. K-means is best suited for spherical clusters.

3.5 Support Vector Machines (SVMs)

SVMs attempt to find the best boundary (or hyperplane) that separates data into different classes, maximizing the margin between data points of different categories. Vladimir Vapnik and Alexey Chervonenkis first introduced the theory of Support Vector Machines (SVMs) in the 1960s, but the modern version of the algorithm was developed in 1992 by Bernhard E. Boser, Isabelle Guyon, and Vladimir Vapnik. The seminal work A Training Algorithm for Optimal Margin Classifiers was presented at the Fifth Annual Workshop on Computational Learning Theory (COLT) in 1992.

- **Computation:** Can be computationally expensive, particularly with large datasets, as it requires solving a quadratic optimization problem.
- **Model Size:** The model's size depends on the number of support vectors, which can increase with large datasets.
- **Speed:** Training can be slow, especially with complex kernels, but prediction is generally fast.
- **Metrics:** Accuracy, precision, recall, and F1 score are standard for classification. SVMs are ideal when there is a clear margin of separation between classes.

For a more in-depth explanation, refer to "Support Vector Machines Explained" (Scholkopf and Smola, 2021), which details the computational and theoretical aspects of SVMs.

3.6 Limitations of Traditional ML algorithms

Although traditional machine learning has been the workhorse of AI for decades, it is not without limitations. Reliance on feature engineering means that the quality of the model is often limited by the expertise of the practitioner. In fields like image recognition and natural language processing, traditional algorithms struggle because

manually identifying useful features is incredibly complex. Enter deep learning, a transformative approach that addresses these challenges.

3.7 Deep Learning Methods

Deep learning is inspired by the structure of the human brain and its neural networks. Unlike traditional ML algorithms, deep learning automatically discovers representations from raw data. This leap in capability is enabled by neural networks with many layers, hence the term ”deep” learning. The success of deep learning is largely attributed to the availability of large datasets, powerful computing hardware, and advancements in optimization techniques. These ideas will be covered in later sections of this chapter. For an in-depth overview of neural networks, see ”Deep Learning” by Ian Goodfellow, Yoshua Bengio, and Aaron Courville (2016) .

3.7.1 Artificial Neural Networks (ANN)

At the heart of deep learning is the neural network, composed of layers of interconnected nodes (neurons) that process data through weighted connections. ANNs are the simplest form of neural networks. They consist of an input layer, hidden layers, and an output layer. Each layer transforms the input, passing information forward and adjusting weights to minimize errors. Warren McCulloch and Walter Pitts developed the first artificial neuron model, laying the foundation for ANNs. This model was inspired by how biological neurons might function. Their paper A Logical Calculus of Ideas Immanent in Nervous Activity was published in Bulletin of Mathematical Biophysics in 1943.

Basic ANNs with a small number of layers and neurons can be adapted for TinyML, though deep networks with many layers are impractical without significant optimization. ANN models can be compressed to reduce their size, but their performance is often limited compared to simpler traditional ML models.

- **Computation:** Computationally intensive, especially as the number of layers and neurons increases.
- **Model Size:** Moderate to large depending on the number of layers and parameters. Memory requirements grow as the network depth and width increase.
- **Speed:** Slower to train compared to traditional models but can be fast for prediction if optimized.
- **Metrics:** Accuracy, precision, and recall are commonly used for classification, while MSE is used for regression tasks. ANNs excel in scenarios where manual feature engineering is difficult.

3.7.2 Convolutional Neural Networks (CNN)

A CNN uses a series of filters that slide across the input image, extracting features such as lines and corners. These features are passed through layers of pooling and activation functions to learn hierarchical representations from simple shapes to complex objects. CNNs are designed specifically for processing structured grid-like data, such as images. CNNs use convolutional layers that automatically detect patterns like edges, textures, and shapes. They have revolutionized fields such as image recognition, object detection, and even video analysis.

Kunihiko Fukushima introduced the concept of a CNN in 1980 with his Neocognitron model, designed to recognize visual patterns. This architecture was further developed by Yann LeCun in the 1990s with the introduction of CNNs for digit recognition. LeCun's seminal work Gradient-Based Learning Applied to Document Recognition was published in the Proceedings of the IEEE in 1998.

Although standard CNNs are too computationally demanding for most TinyML applications, lightweight versions like MobileNet and SqueezeNet have been developed specifically for resource-constrained environments. These models use fewer parameters and computational resources, which makes them more suitable for TinyML.

- **Computation:** Highly computationally intensive due to convolution operations, especially with deep architectures like VGG or ResNet.
- **Model Size:** Large, as each layer adds numerous parameters. CNNs require substantial memory, particularly for high-resolution images.
- **Speed:** Slower to train but relatively fast for inference if implemented on specialized hardware like GPUs.
- **Metrics:** Accuracy, precision and IoU (Intersection over Union) for tasks such as image classification or object detection. CNNs are best suited for vision tasks and large-scale image datasets.

3.7.3 Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM)

RNNs are used for sequential data, where previous outputs inform future predictions, making them ideal for tasks like time series forecasting and natural language processing. LSTM is a type of RNN that addresses the vanishing gradient problem by using memory cells to store and retain information over longer sequences. It is effective for tasks requiring long-term dependencies, such as speech recognition and language translation. RNNs and LSTMs, due to their sequential processing and high memory requirements, are generally not well-suited for TinyML. However, small RNN models optimized through pruning or quantization can be used for limited tasks such as basic time series forecasting or voice recognition.

RNNs were introduced by David Rumelhart, Geoffrey Hinton, and Ronald J. Williams in 1986. Later, Sepp Hochreiter and Jürgen Schmidhuber developed LSTM networks to address the vanishing gradient problem in RNNs in 1997. The original RNN work appeared in Learning Representations by Back-Propagating Errors

(1986), while the LSTM paper, Long Short-Term Memory, was published in Neural Computation (1997).

- **Computation:** Computationally intensive due to the need to process sequences of data.
- **Model Size:** Large, especially with long sequences, as the model must retain information over time.
- **Speed:** Slower than CNNs due to sequential processing, though faster variants like GRU improve performance.
- **Metrics:** Accuracy and BLEU score for NLP tasks, and MSE for time-series prediction. LSTMs are preferred when capturing long-range dependencies in sequential data.

3.7.4 Transformer Networks

Transformers use self-attention mechanisms to process sequential data in parallel, allowing them to capture long-range dependencies without relying on recurrence. They have revolutionized tasks in natural language processing and are the basis for models like BERT and GPT. The Transformer architecture was introduced by Vaswani et al. in 2017. This new model revolutionized natural language processing by introducing the attention mechanism, allowing for parallelization during training, unlike RNNs. The paper Attention is All You Need was published at the Advances in Neural Information Processing Systems (NeurIPS) conference in 2017.

Transformers, such as GPT and BERT, are known for their heavy computational and memory demands. As a result, they are not practical for TinyML applications unless significantly scaled down. Some research has been conducted on simplifying transformers for constrained environments, but the trade-offs in performance make them less appealing for TinyML.

3.8 Emerging ML methods: Tsetlin Machines

Deep learning introduces complexity in architectures for ML applications. The trend seems to be added complexity with each new algorithm i.e. fully connected layers to convolution to transformers. The recently proposed Tsetlin Machine (TM) [6] offers one potential alternative to this direction of development. The TM is an intrinsically logic based algorithm that relies only on bitwise operations and integer addition for inference. This makes them extremely well suited for TinyML applications in terms of both compute and memory.

The core components of the TM are presented in Fig. 3 and a more detailed description is offered in [6] and [44]. Following Fig. 3, the TM must convert the inputs into Boolean Literals for the inference computation. The example shows how an input datapoint is converted into these Boolean Literals. This process is referred to as *Booleanization*. In the example in Fig 3, the Booleanization is a conversion

of the integer value into its binary representation. However, other approaches are available based on the precision that the user would like to use for their learning problem. This is discussed further in [45].

Each Boolean Literal has its own learning element called the Tsetlin Automata (TA). The TA is a finite state machine with two actions `Include = 1` and `Exclude = 0`. Each TA will have an equal number of states corresponding to each action. The clause is the bitwise operation that links each TA to its respective Boolean Literal to generate a 1 bit clause output.

The 1 bit clause output forms the basis of the TM inference architecture. For a multi-class classification problem, each clause in each class contributes a vote to form a class sum. The clause may have either a positive polarity ("multiply" the clause output by +1) or a negative polarity ("multiply" the clause output by -1). The class sum from each of these clause votes will go through an argmax function to produce the final classification.

- **Computation:** The use of bitwise operations for the clause computation and the integer-based class sum compute makes the TM one of the simplest algorithms for computation.
- **Model Size:** Relatively large, the TM typically requires a large number of clauses to solve complex problems. Therefore, memory compression approaches are paramount to porting TMs to TinyML devices.
- **Speed:** The simplicity of the operations can be exploited to yield fast speeds.
- **Metrics:** Test accuracy (%) is the only important metric as precision is set intrinsically to a single bit.

Summarizing: There are limitations to all mentioned algorithms that prevent them from being easily deployed on the edge. The main issues are the floating-point computations and the memory footprints. Addressing these two points is the central idea for the co-design of algorithms for TinyML and is explored in the following section.

4 Co-Design

This section condenses the theoretical concepts presented by seminal papers dealing with co-design of ML algorithms. The following overview summarizes the most effective concepts.

Overview

This section is concerned with how the trained ML models from the previous section can be *compressed* or *distilled* into smaller and less compute intensive versions while retaining as much as the original trained model's performance characteristics. These methods include *quantization* of the precision of learning elements, *pruning* of redundant or insignificant components for *compression* into more memory-frugal data structures suitable for edge nodes.

4.1 Compression

The compression of models is central to fitting a trained model into the TinyML device. Compression typically involves reduction of the precision of the learning elements and data (referred to as Quantization), removal of redundancy or less important model components (referred to as Pruning) or exploiting model sparsity to store the model in smaller data structures. These three ideas are discussed next.

4.1.1 Quantization

Quantization involves reducing the precision of the elements used in the parameters of a model. Instead of storing and high-precision 32-bit floating point numbers (FP32), we use 16-bit or 8-bit integers (INT8). This drastically reduces the model's memory needs and accelerates the inference computations as the compute intensity is now reduced. Of course, there is a trade-off: lowering precision can sometimes reduce accuracy, but with careful tuning, the impact is often minimal. This process can happen after (post-training quantization) the model training or during (quantization aware training). Both of these are now considered in turn:

Post-training quantization: Once the model is trained, its parameters such as weights and activations are reduced to a lower precision. Typically this comes with a small reduction in accuracy. Contoli et al.[46] provide an insightful example of Post-Training Quantization, particularly in the context of Human Activity Recognition tasks on low-power devices. The impact was examined for three compression techniques lite conversion, dynamic quantization, and full-integer quantization on accuracy and energy consumption when deploying models on an ESP32 microcontroller. Their results show that dynamic quantization offers the best balance, with

negligible accuracy loss (less than 0.1%), while achieving up to 30% energy savings, as shown in Fig. 4. On the other hand, full-integer quantization slightly outperforms dynamic quantization in energy savings but causes a more significant 2-3% drop in accuracy, particularly for larger models. Moreover, the accuracy changes as network complexity increases (from C1 to C8 for 1D CNN and CL1 to CL6 for 1D CNN LSTM), demonstrating that dynamic quantization retains high accuracy even for complex models.

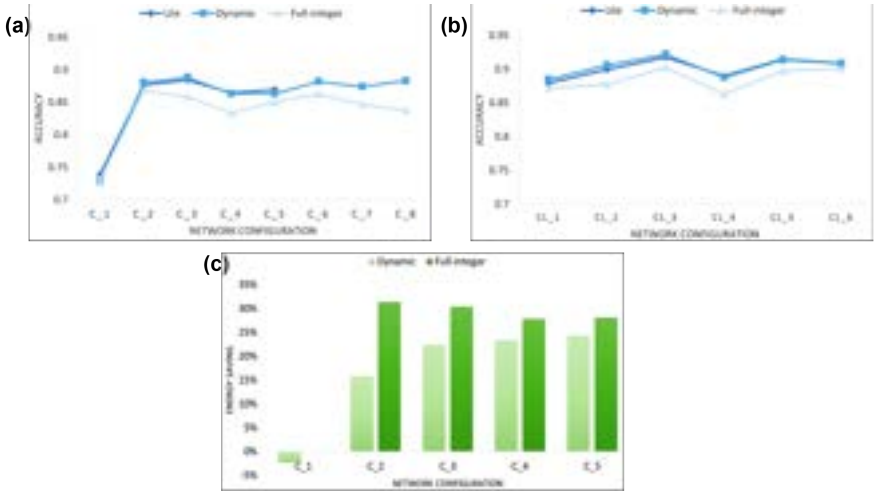
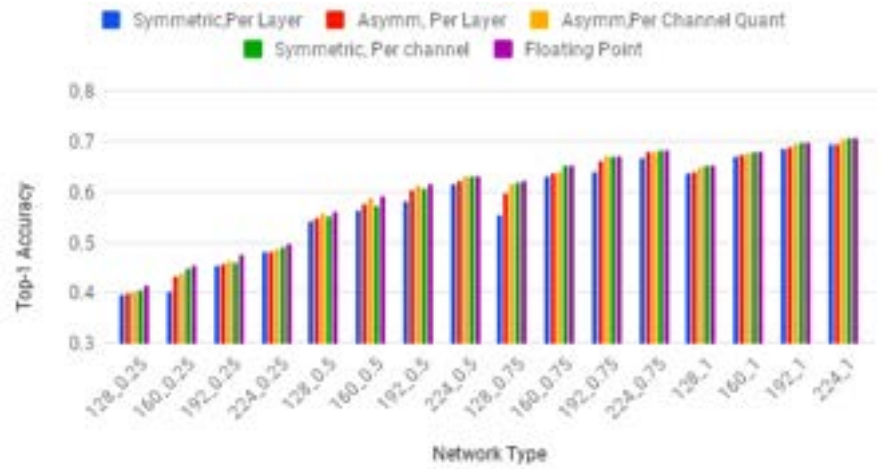


Fig. 4: (a) and (b) Classification accuracy of the 1D CNN (one-dimensional convolutional neural network) and 1D CNN LSTM (long short-term memory) models as a function of network complexity. (c) Classification accuracy of the 1D CNN (one-dimensional convolutional neural network) and 1D CNN LSTM (long short-term memory) models as a function of network complexity. Reprint from [46]

Quantization-aware training (QAT): in this approach, the model is trained with quantization in mind, simulating the lower precision during training. This helps the model better adapt to being compressed, resulting in less loss of accuracy compared to post-training quantization. Krishnamoorth [47] used QAT, where quantization is incorporated during the training process, resulting in higher accuracy compared to post-training quantization. QAT allows both weights and activations to be quantized during training, and the model learns to adjust to the quantization-induced noise. QAT reduced the accuracy gap to floating-point models to within 1% at 8-bit precision, and at 4-bit precision, the accuracy loss ranges from 2-10%, depending on network size as seen in Fig. 5. Additionally, the results demonstrate that QAT offers significant improvements at lower bitwidths, making it particularly effective for resource-constrained environments.



Network	Asymmetric, per-layer (PTQ)	Symmetric, per-channel (PTQ)	Asymmetric, per-layer (QAT)	Symmetric, per-channel (QAT)	Floating Point
Mobilenet-v1_1_224	0.001	0.591	0.70	0.707	0.709
Mobilenet-v2_1_224	0.001	0.698	0.709	0.711	0.719
Nasnet-Mobile	0.722	0.721	0.73	0.73	0.74
Mobilenet-v2_1.4_224	0.004	0.74	0.735	0.745	0.749
Inception-v3	0.78	0.78	0.78	0.78	0.78
Resnet-v1_50	0.75	0.751	0.75	0.75	0.752
Resnet-v2_50	0.75	0.75	0.75	0.75	0.756
Resnet-v1_152	0.766	0.762	0.765	0.762	0.768
Resnet-v2_152	0.761	0.76	0.76	0.76	0.778

PTQ: Post-Training Quantization. QAT: Quantization-Aware Training.

Fig. 5: The Comparison of quantization-aware training schemes for Mobilenet-V1. The graph shows how QAT closes the accuracy gap to floating-point performance, even at 4-bit quantization. The table: comparison of top-1 accuracy for different quantization schemes shows that QAT reduces the accuracy gap compared to PTQ, particularly at lower precisions. Reprint from [47].

Dynamic quantization: the model maintains high precision when stored but uses lower precision computations during inference. It is a balance between performance and accuracy. For TinyML, quantization is one of the go-to strategies because it can dramatically reduce the model's footprint, making it feasible to run on small devices without much sacrifice in performance. Angelo, et al.[48] proposed the DARKSIDE architecture for Extreme Edge On Chip DNN Inference and Training was proposed. Employing the dynamic quantization technique achieved an impressive energy efficiency of 835 GOPS/W for 2/4/8-bit integer kernels with peak performance of 65 GFLOPS and 300 GFLOPS/W for FP16 workloads with 18.2 GFLOPS. When applied to the Bottleneck layer of MobileNetV2, the system shows a 1.85× improvement in performance using the RVNN cores (RISC-V Neural Network) with M&L (fused MAC-load instruction) and the specialized Depthwise Engine. These

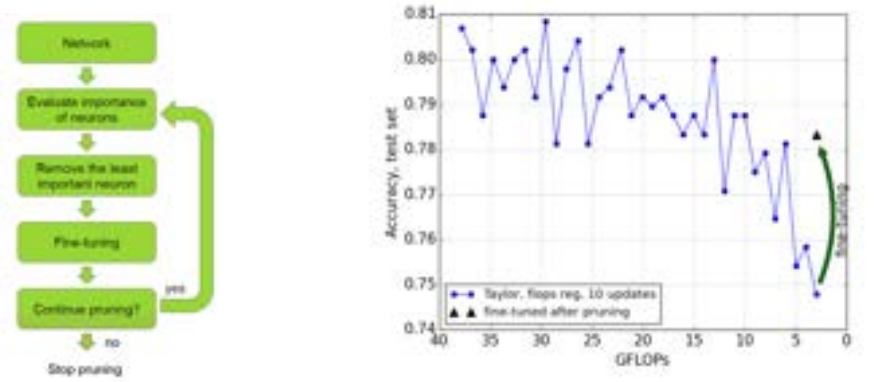
optimizations, along with the DataMover for efficient data organizing, significantly reduce execution time and boost energy efficiency.

4.1.2 Pruning

This technique involves removing less important weights or neurons from a neural network, which can lead to significant reductions in the model's memory footprint and computational requirements and minimal impact on the model's output. This can be done in various ways.

Weight pruning: focuses on removing individual weights from the model. Weights that have minimal impact on the output are identified and pruned. Molchanov et al.[49] proposed an approach to pruning convolutional kernels in neural networks to enhance inference efficiency. The main focus is on pruning entire feature maps rather than individual weights, and the method interleaves greedy criteria-based pruning with fine-tuning by back-propagation using Taylor expansion to approximate the cost function. This ensures the pruned networks retain good generalization abilities. Their method demonstrated that pruning could be effectively integrated into the training process, resulting in models that are both lightweight and efficient. The method starts by fine-tuning the network, followed by iterative pruning and further fine-tuning until the desired balance between accuracy and efficiency is achieved. Taylor expansion, which is based on first-order gradient information, is used to approximate the impact of pruning on the cost function. The proposed pruning outperforms other criteria, such as weight magnitude pruning, activation statistics, Optimal Brain Damage (OBD), and mutual information, in terms of maintaining accuracy while reducing the number of parameters and the reduced GFLOPs (giga floating point operations per second). It encapsulates the effectiveness of the pruning method on a real-world transfer learning task. Moreover, A recurrent 3D-CNN was pruned for dynamic hand gesture recognition using the Taylor criterion, as shown in Fig. 6. The pruning process reduces the required GFLOPs from 37.8 to 3.0, while the accuracy drops by 2.5%, which is largely recovered with additional fine-tuning. Significant speedups were achieved using pruned networks on hardware. For example, with the pruned AlexNet / Flowers-102, VGG-16 / ImageNet a 2.1x to 3.4x speedup where achieved compared with various GPUs and CPUs.

Neuron Pruning: This technique involves removing entire neurons from the network. As a result, this will lead to a more substantial reduction in model size compared to weight pruning. For example, Chen et al.[50] explored how pruning can be effectively applied to Vision Transformers (ViTs) to reduce computational costs while maintaining or even improving model accuracy. The used pruning techniques are one-shot weight magnitude pruning and gradually magnitude pruning, which seamlessly integrates pruning into the training process as shown in Fig. 7a. Through sparse Vision Transformer Exploration (SViTE), the unimportant parameters were dynamically pruned during training, reducing the model size and FLOPs, as shown in Fig. 7b. An extended version, S2ViTE, introduces structured sparsity by pruning self-attention heads, making the model more hardware-friendly. Furthermore, SViTE+



(a) The network pruning approach

(b) Pruning of a recurrent 3D-CNN for dynamic hand gesture recognition.

Fig. 6: The Weight pruning process [49].

co-explores data and architecture sparsity by using a token selection mechanism that identifies the most important image patches, allowing the model to operate more efficiently. A prim example of the effectiveness of pruning is seen in SViTE+-Small, which, at 50% unstructured sparsity combined with 5% data sparsity, achieves 49.32% FLOPs savings and even improves the top-1 accuracy to 80.18%. This result showcases that by selectively pruning less important components, the model not only becomes more efficient but can also slightly enhance its performance, highlighting the power of pruning as an essential technique for modern machine learning models. The models used in the study are based on DeiT (Data-efficient Image Transformers), which are specialized ViTs designed for image classification with enhanced training efficiency.

Structured Pruning: Instead of pruning individual weights, structured pruning removes entire structures, such as filters or channels, from convolutional layers. This method can lead to more efficient implementations on hardware. For instance, Zhang et al.[51] discussed AKECP (Adaptive Knowledge Extraction for Channel Pruning), a method designed to efficiently prune DNNs by extracting effective knowledge from feature maps (FM). AKECP addresses the limitations of previous FM-based pruning techniques by using fewer images, sometimes just one, to establish pruning criteria, significantly improving the speed and efficiency of pruning. It prunes all layers simultaneously, avoiding the slow layer-by-layer process, and achieves substantial compression without significant accuracy loss. For instance, on ResNet-110, it reduces parameters by 59.9% and FLOPs by 59.8% on CIFAR-10, and on ResNet-50, it cuts 40.5% of memory and reduces FLOPs by 44.1% on ImageNet with only a 0.32% drop in Top-1 accuracy.

Dynamic Pruning: adapts the pruning process based on the model's performance during training or inference. For example, Pronoet et al.[52], showcased dynamic

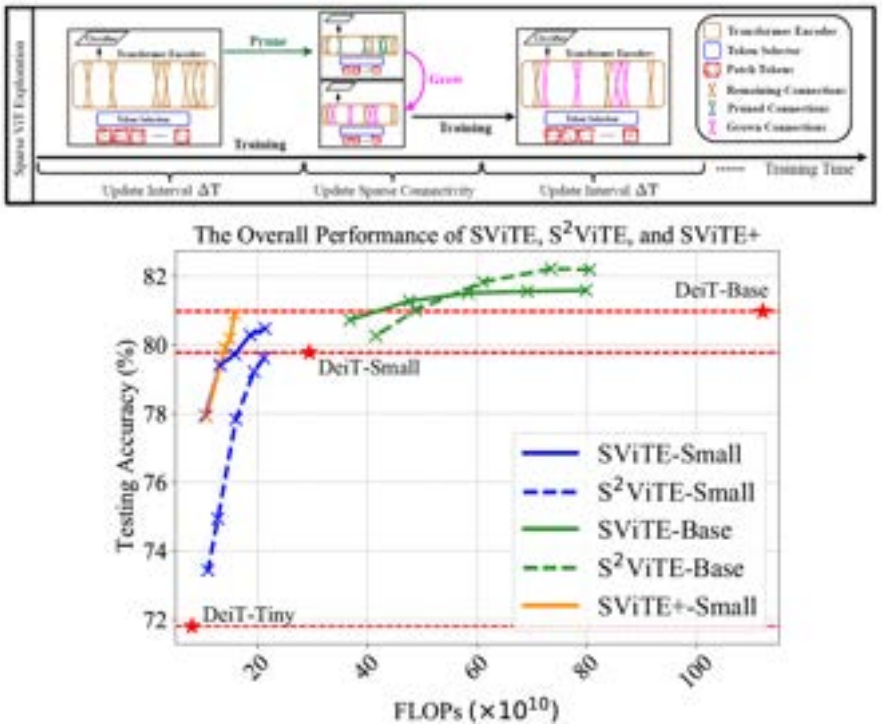


Fig. 7: (a)SViTE process to explore critical sparse connectivities, (b)Top-1 accuracy vs. FLOPs trade-off for different sparse models (SViTE, S2ViTE, SViTE+) on ImageNet, demonstrating the efficiency improvements through pruning [50].

pruning in the context of IoT applications, where models were pruned based on real-time performance metrics. The dynamic pruning utilized a novel Multiply-And-Max/min (MAM) map-reduce neuron paradigm, enabling substantial pruning of DNNs with minimal accuracy loss. Unlike traditional MAC neurons, MAM neurons enhance interconnection sparsity, allowing for much more aggressive pruning. The MAMs demonstrated effectiveness on several DNN architectures, including AlexNet, VGG-16, and Vision Transformers (ViT-B/16), using datasets like CIFAR-10, CIFAR-100, and ImageNet-1K. Notably, MAM neurons retained 98% of the model's accuracy while reducing interconnections to just 0.04% on the ViT-B/16 model, compared to MAC layers requiring 56.16% of interconnections to achieve the same accuracy. Moreover, using AlexNet, MAM-based models maintained 87.34% accuracy on CIFAR-10 with only 0.03% remaining weights, while MAC-based models needed 0.48% of the original weights. For CIFAR-100, MAM maintained 62.52% accuracy with 0.26% weights kept compared with 25.01% weights kept for MAC. The MAM's ability to dramatically reduce the model size and FLOPs while preserv-

ing performance makes it a powerful technique for dynamic pruning. Mathematically ,as shown in Fig. 8, the output z_i of the MAM neuron for the i-th neuron would be:

$$z_i = \max(v_{i1}, v_{i2}, \ldots, v_{iN}) + \min(v_{i1}, v_{i2}, \ldots, v_{iN}) + b_i$$

Where v_{ij} is the weighted input (the product of the j^{th} input x_j and the corresponding weight w_{ij}). $\max(v_{i1}, \ldots, v_{iN})$ is the largest value from the weighted inputs, $\min(v_{i1}, \ldots, v_{iN})$ is the smallest value from the weighted inputs and b_i is the bias term added to the final output for the i-th neuron.

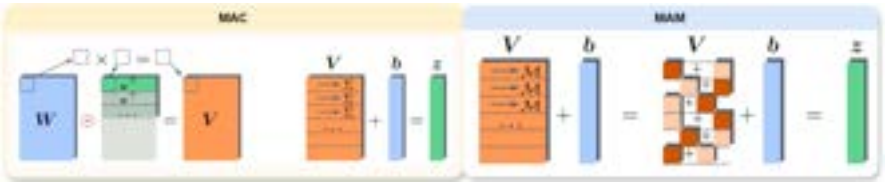


Fig. 8: The MAC and the MAM neuron paradigm in reducing the number of interconnections.

Iterative Pruning: This method involves repeatedly pruning the model and then fine-tuning it to recover any lost accuracy. Disabato et al.[53] highlighted the importance of iterative pruning in TinyML, where models are pruned and fine-tuned incrementally to adapt to the limited resources of embedded systems. The technique leverages transfer learning combined with a kNN classifier, allowing on-device learning without the need for off-device training. This incremental approach supports gradual model adjustments, making the system more adaptable to new data and environmental changes. Their experiments in image classification and speech command recognition demonstrated that as more samples are introduced, classification accuracy improves iteratively. For instance, in the image classification scenario, the kNN classifier's accuracy plateaus after 20-30 samples per class, while the Support Vector Machine (SVM) model performs slightly better with a gap of 0.05–0.10 in accuracy where the SVM classifier used as a comparison is not incremental. Although SVM performs better, it requires more complex training, which is impractical to perform the training phase for embedded devices such as MCUs. Furthermore, by employing dimensionality reduction, computation time is significantly reduced, especially in the feature extraction and classification stages. where, the feature extraction on the STM32F7 microcontroller took 2.099 seconds per image, compared to 0.154 seconds on the more powerful Raspberry Pi 3B+. Similarly, kNN inference time increased from 0.004 seconds on the Raspberry Pi to 0.009 seconds on the STM32F7 with 10 neighbors, showcasing the effectiveness of pruning techniques even in constrained hardware environments.

4.1.3 Sparsity based compression for Tsetlin Machines

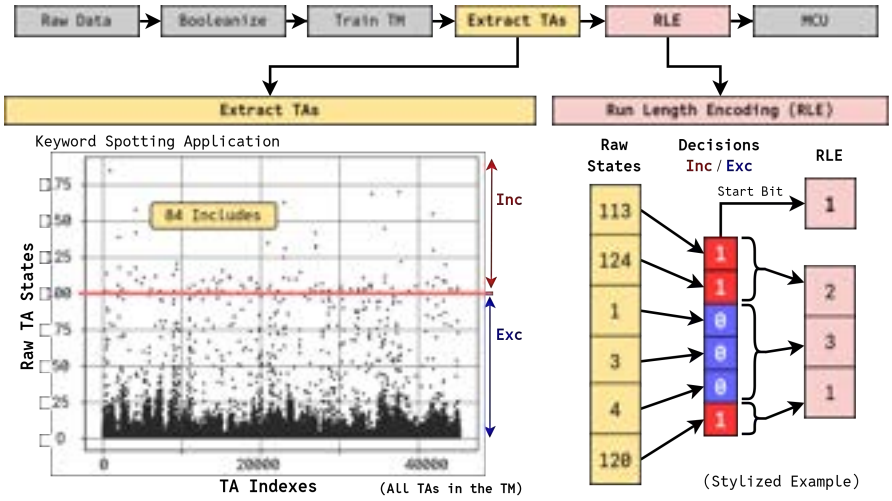


Fig. 9: One possible approach to exploiting sparsity in Tsetlin Machine models. There is a large number of exclude TAs vs include TAs. This allows trained TM models to be represented in compressed data structures such as Run Length Encoded arrays. This approach is used by [54].

Certain models have intrinsic sparsity. This is true for Tsetlin Machine where the number of excludes far outnumbers the number of includes, as seen in Fig. 9. This kind of sparsity leads to more than 95% compression of the model and is explored in [54] where these compressed TMs are compared against embedded binary neural networks (eBNN). Notice in the Extract TAs graph for a Keyword Spotting application, there are only 84 includes in a model with 45K TAs. If the TA include and exclude decisions are flattened into a 1D array, there will be large runs of `0`s before a `1` appears. The strategy used in this figure is to use Run Length Encoding (RLE) to compress the large runs of excludes. This style of encoding is suitable for TinyML implementations where the compressed model will be iterated one element of the RLE array at a time. This is very suitable for microcontrollers.

4.1.4 Knowledge Distillation

Knowledge distillation is a powerful model compression technique that involves transferring knowledge from a larger, more complex model (”teacher”) to a smaller, simpler model (”student”). This process allows the student model to achieve per-

formance levels comparable to the teacher model while significantly reducing its size and computational requirements. The fundamental idea behind knowledge distillation is that the teacher model has learned rich representations and decision boundaries during training, which can be effectively utilized to guide the training of the student model, which results in a lightweight model that's efficient enough for TinyML devices, yet still performs admirably in the real world. The process of knowledge distillation generally involves two phases: training the teacher model and training the student model. In the teacher training phase, a high-capacity model, often a deep neural network with multiple layers and parameters, is trained on a dataset to achieve the best performance. The model's complexity allows it to learn complex patterns within the data. In the student training phase, the outputs of the teacher model serve as "soft targets." Instead of relying solely on hard labels (i.e., actual class labels), the student model learns from the probability distributions generated by the teacher model. These soft targets provide richer information about the relationships between different classes, enabling the student model to generalize better. The training process involves minimizing the difference between its outputs and those of the teacher model, typically by employing a loss function that combines traditional cross-entropy loss with a distillation loss, which quantifies the divergence between the teacher's and student's outputs [55]. Shahraki et. al.[56] explored the use of knowledge distillation to create lightweight models suitable for IoT traffic classification. It demonstrated that the student model achieved almost comparable accuracy to the larger teacher model while being more resource-efficient. Specifically, the student model achieved an accuracy of 98.31% on the Cambridge dataset, which is only slightly lower than the teacher model's 98.51%. The size of the student model was 768 KB, a significant reduction compared to the teacher model's 9,060 KB, with fewer parameters and reduced computational cost. The study highlights that knowledge distillation is effective even with reduced training data and shows robust performance when some traffic classes are excluded, emphasizing the value of soft labels for training student models under limited data scenarios.

4.1.5 Weight Clustering

Weight clustering (or weight sharing) is a compression technique that reduces the number of unique weight values of the Neural Network to be stored for efficient compression as well as representation. It groups the weights of a neural network layer into clusters and shares the centroid value of each cluster [57]. The process of weight clustering involves the following:

- **Clustering Weights:** divide the weights of each layer into an X number of clusters, where each weight is allocated to the closed cluster.
- **Weight Sharing:** the centroid of each cluster will be stored in the model rather than saving the individual weights, where all weights that belong to a cluster are replaced by the centroid value. This method reduces the number of unique weight values and allows the model to be compressed without significant loss in performance.

- **Centroids tune:** the cluster of the centroids can be fine-tuned to reduce the error by initializing it randomly, linearly, or density-based. The linear technique yielded superior results. Because large weights are more significant, utilizing linear distribution increases their chances of generating a large centroid.

4.1.6 Filter Bank Design

Filter bank is a crucial aspect of signal processing in TinyML, particularly for applications involving audio, image, and sensor data. A filter bank is a collection of bandpass filters that divide the input signal into multiple components, each representing a specific frequency range. These filters can be used to extract temporal and frequency-domain features, depending on the type of sensor data. This division allows for more efficient processing, analysis of signals, and image compression making filter banks essential for various machine learning tasks in resource-constrained environments. Some types of filter banks are

- Quadrature Mirror Filter Banks (QMF) [58]
- DFT Modulated Filter Banks [59]
- Multi-Band Filtering (MBF) [60]
- Frequency Response Masking (FRM) [61]
- Non-Uniform Filter Banks [62]
- Gammatone Filter Banks [63]
- Wavelet Packet Transform Filter Banks [3]
- Adaptive Filter Banks [64]

5 Implementation

Selecting the *right* existing hardware... or developing *new* hardware

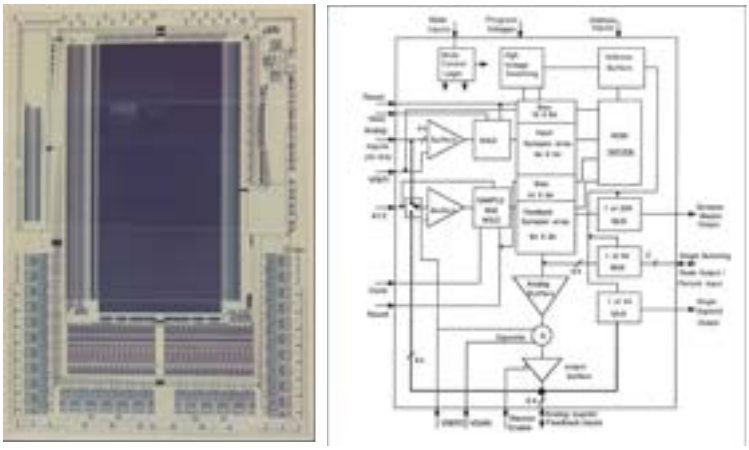


Fig. 10: The Intel ETANN 80170 is widely regarded as the first ML accelerator implementation (1989). ETANN stands for Electrically Trainable *Analog* Neural Network. The block diagram shows that this accelerator only incorporated 64 neurons - truly TinyML! (see [65] for more details on how the US Naval Air Warfare Center used this implementation for missile seeker applications)

Overview

The key focus of this chapter is about taking the optimized model described in the previous section and understanding the rationale for *mapping* it to the most suitable the hardware implementation. Following the decision process presented in Fig. 11, the core *metrics* are derived that form the direction for whether the model is deployed on generalized hardware such as CPUs or GPUs or whether specialized hardware is required (see Fig. 10)[1]. CPU-based implementations have constraints on hardware resources and development support that should be considered. Specialized hardware is underpinned by core design principles that emerge from the question of whether a system is bounded by compute or memory. Ultimately, the specialized implementation is either a custom ASIC, an FPGA, or a Tensor Processing Unit (TPU) [69]. An exploration of recent research works and commercial platforms offers further insight into the direction and trends for TinyML *implementation*.

[1] It is assumed for this section that the reader has some notion already about computer architecture and how CPUs, FPGAs and GPUs work. Otherwise, a quick refresher on the latest functionality and design trends in CPUs and GPUs can be seen here: [66]. Harris and Harris offer a good summary on the fundamentals [67]. For FPGAs, [68] presents a detailed guide to development using Xilinx (AMD) platforms.

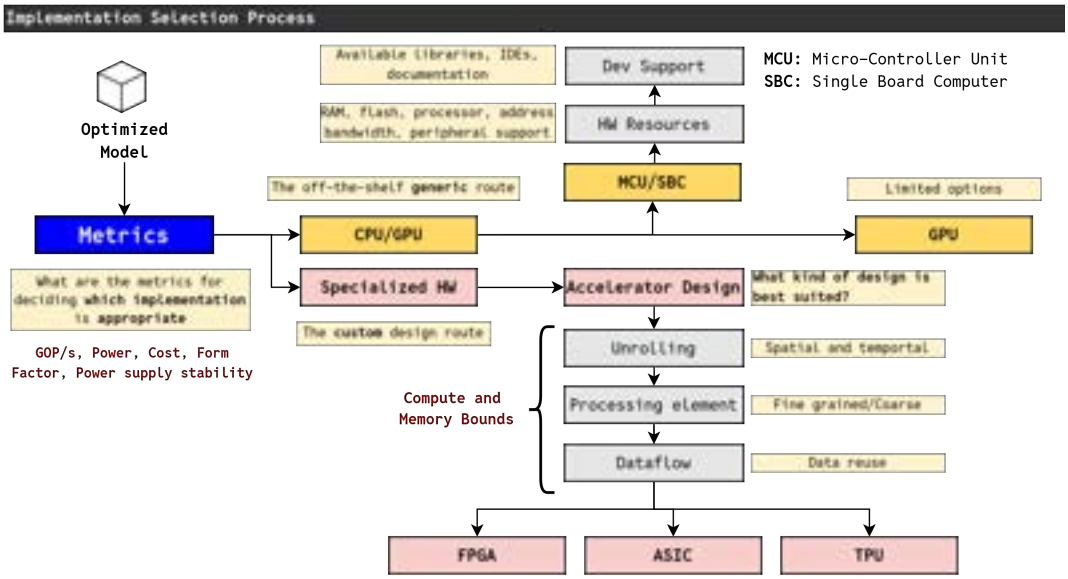


Fig. 11: Visual overview of this section. The focus is selecting suitable hardware for the optimized model. This can be either a general off-the-shelf CPU or GPU or a more specialized architecture.

5.1 *Metrics* Underpin Implementation Choice!

The new metrics: Before discussing the different implementations that the ML algorithm can be deployed on, it is worth understanding a bigger picture of what the implementation is supposed to offer. In many ways, implementation is the convergence point of the TinyML pipeline, it is the culmination of the previous two sections of algorithm and co-design now mapped to a hardware platform. There are certain metrics that are important for TinyML - from the learning efficacy perspective there is accuracy - e.g. test accuracy or F1 score. It could also be latency and model size.

Fig. 11 demonstrates that implementations begin with metrics. When mapping the algorithm to a platform, newer metrics emerge in addition to what was seen in the previous sections; these metrics are concerned with how the hardware platform can leverage better energy efficiency from its algorithm workload (i.e. the inference computation using the optimized model). The cost of and the development time required for implementation as well as the form factor (size) also play a crucial part.

For example, ASIC implementation is more costly and time consuming than developing on a FPGA.However the FPGA designs can only be mapped onto existing fabrics and typically of a larger form factor and are less energy efficient. Similarly CPU good generalization but there is a plethora of choice when it comes to MCUs and Single Board Computers (SBCs). GPUs typically offer better throughput than CPUs for tensor based models but at the cost of higher power consumption.

Fig. 11 organizes the implementation choice into two routes. Either a generic route and a specialized route. The reminder of this section will examine both these route in terms of decision decisions, available platforms, recent research and most importantly how they address the implementation metrics.

The two routes that can be taken for TinyML implementation:

- **The generic route:** Deploy the optimized algorithm on a existing piece of hardware like a RISC based CPU (typically in a MCU development board), a single board computer (SBC) - or small GPU.
- **The specialized hardware route:** Design an *accelerator*. This is a custom hardware implementation to deploy the optimized algorithm - this could be an ASIC or FPGA or a commercially available TPU.

Making the decision between the routes: The key to making the decision of which path to take is determined by how much computation should be completed within a given power budget for the application.

The Big Picture: Fig. 12 shows three graphs that explain how custom implementations should be measured, not only for TinyML implementations but across the range of all ML accelerators. These implementations have focused solely on Neural Network based algorithms. The axis of the graph show the two main metrics: Number of Giga Operations (GOPs) and the Power (W). These graphs are from the Nanoscale Integrated Circuits and Systems Lab, Department of Electronic Engineering, Tsinghua University, Energy Efficient Computing Group (NCIS-EFC). They show the most up-to-date hardware custom platforms from industry and research as reported in IEEE Transactions on Circuits and Systems I, Journal of Solid State Circuits and other prestigious publication channels. The key point to note is how the GOPs/W pareto frontiers constrain the different implementation platform types. For example the ASIC implementations are more power efficient while the FPGA ans GPU can achieve greater GOPs at the expense of more power - going well beyond what is appropriate for TinyML [2].

Therefore, choice of implementation is centred around how to maximise the GOP/s within the power budget of the TinyML application. One important finding from the top graph is the groupings formed by the different hardware implementations. ASICs offer the most energy efficient option while FPGAs, CPUs, and GPUs take up more power. Additional nuances to this will be explored later.

The two graphs below the Platform graph in Fig. 12 shows the different co-design techniques that have been used by each of the implementations. Many have utilized some degree of quantization but far fewer have looked at sparsity. For the sparsity key, **A** refers to activation function and **W** refers to the weights.

[2] These graphs are interactive - a specific point of interest (i.e. a accelerator implementation) can be examined if interested - see here: `https://nicsefc.ee.tsinghua.edu.cn/project.html`. These accelerators are not limited to TinyML applications but are useful for demonstrating the overarching trend

A caveat when making direct comparisons between implementations: Notice the GOP/s in the top graph do not account for precision. The precision of the model can play a huge part in the number of operations that can be performed and the power budget as seen in the second graph (this will be explored in detail later). Therefore, it is important to look at GOP/s and GOPs/W with respect to *precision*.

The next two subsections explore the two routes seen in Fig. 11 of either a generic platform or a specialized platform. The focus of these subsections is to understand how the specific hardware components in each platform type (CPU, GPU, ASIC, and FPGA) can be harnessed to optimize one or more of the metrics encountered so far. But also to understand how intrinsic properties of both the algorithm and the hardware can also lead to distinct clusters as seen in Fig. 12.

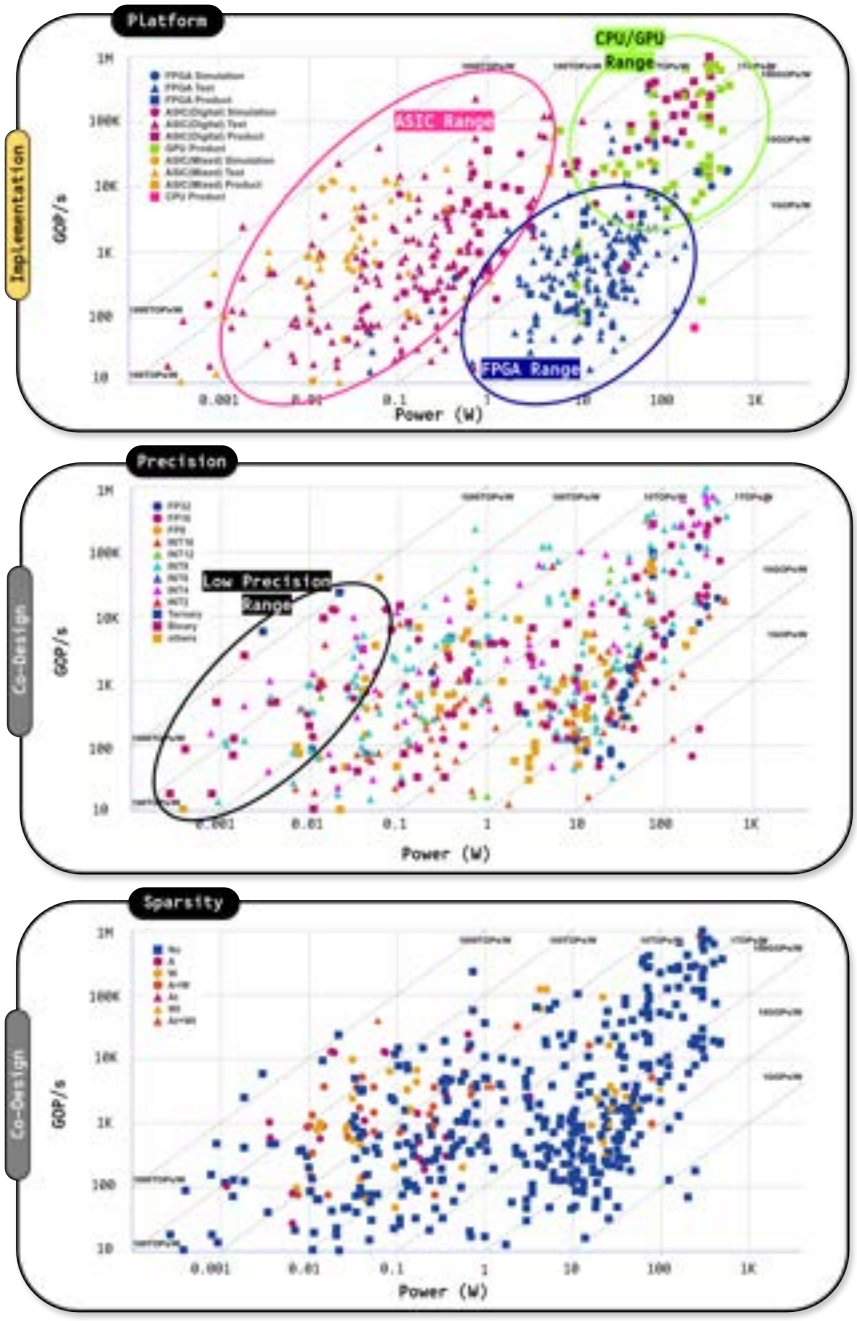


Fig. 12: Neural Network implementation map adapted from data presented in [70].

5.2 The Specialized HW Route

Taxonomy and design principles for accelerators

This subsection explores the specialized hardware route presented in Fig. 11. The rationale for going down this route of custom hardware is the freedom in design capability when mapping the optimized model to either an ASIC, FPGA or TPU. Therefore, it is important to understand the core principles used in this mapping (rolling, processing element design, and data flow)[3].

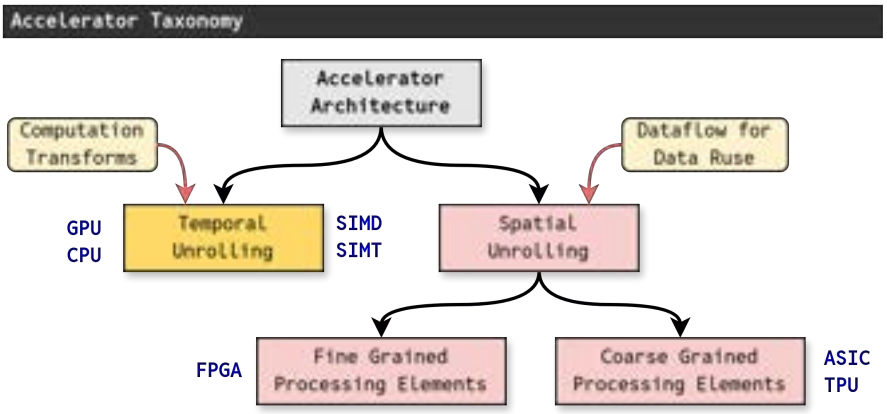


Fig. 13: Generalizing accelerator design approaches into a taxonomy. We will refer to this diagram frequently throughout this section.

5.2.1 The Taxonomy for Accelerator Designs

At the heart of the ML algorithm is a computation. Typically for Neural Network based approaches these are General Matrix Multiplications (GEMM). However, any algorithm that contains `for` loops are also applicable (e.g. TMs). The role of the accelerator is to maximise the GOPs through highly parallel structures. However, these structures cannot be infinitely parallel as there are always restrictions in compute and memory. Therefore, the design must carefully balance the memory and compute *bounds* of the platform. For the remainder of this section, a particular focus is placed on DNNs. The GEMM operations in DNNs expose opportunities to discuss all the main design decisions expressed in Fig. 11. It is worth noting that the taxonomy is extended to all ML accelerators, not just for TinyML implementations.

Spatial and Temporal Unrolling: Fig. 13 shows how accelerator design approaches can be organized into a taxonomy. The first branch of the taxonomy occurs

[3] Given the plethora of options that are available in this area, only the most state-of-the-art is considered when reviewing implementations, and only the most widely used commercial platforms are selected for review.

between *Temporal Unrolling* and *Spatial Unrolling*. Typically, accelerators will use a combination of both temporal and spatial unrolling. The choice of how much of each to use depends on whether the implementation is more compute-bound or memory-bound. These ideas will form the core focus of this subsection.

Temporal unrolling is common in CPU and GPU architectures as well where they may use Single Instruction Multiple Data (SIMD) or Single Instruction Multiple Thread (SIMT) [71]. Typically implementations like this are very resource heavy e.g. many ALUs and not so appropriate for TinyML. In comparison spatially unrolled architectures are more common for DNN based TinyML implementations. They key design principles for each unrolling is either computational transforms (loop reordering or breaking) or maximising data reuse [71].

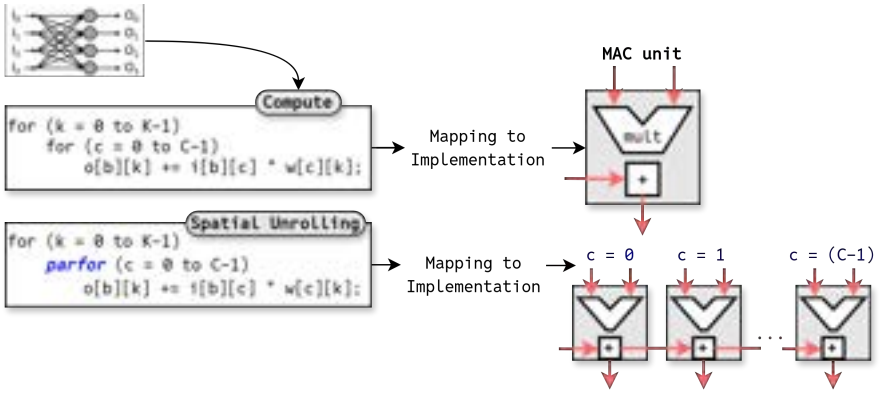


Fig. 14: Visualizing spatial unrolling. The processing elements here are the MAC units. The first nest `for` loop box (Compute) represents a temporal unrolling, the same MAC unit is used again and again for all the iterations of the loop.

Focusing on spatial unrolling, this approach involves the parallel execution of multiple operations by allocating distinct hardware resources (such as multiple processing units (PEs)) for each operation. Instead of reusing a single computational unit, multiple units are deployed simultaneously to perform the same or different tasks in parallel. This is a key strategy in compute-bound systems, where the bottleneck is the processing capability rather than memory access. By increasing the number of PEs, accelerators can achieve higher throughput, executing multiple operations in parallel. In Fig. 14, the `for` loop in the first compute box is replaced with a `parfor` loop. The compute here represents a fully connected layer; however, any loop iteration compute applies here. In mapping terms for this compute, it is the equivalent of replacing a single MAC unit that is iterated for the two nest for loops with a parallel version. Note that this `parafor` has unrolled the full loop, however this does not have to be the case. The amount of rolling possible depends on the hardware resources available.

Compute Bounded Implementations: In compute-bound systems, the bottleneck is the availability of compute resources. Here, memory access is fast enough to keep up with computational demands, and the performance is limited by the number of operations the hardware can perform. Spatial unrolling is ideal in these scenarios because it maximizes parallelism, allowing more computations to occur simultaneously and thus speeding up execution.

Memory bounded Implementations: In these systems, the performance bottleneck lies in how quickly data can be fetched from memory. If the bandwidth or latency of the memory is the limiting factor, increasing computational resources (as in spatial unrolling) will not yield significant performance improvements, as the computational units will often be idle, waiting for data. Temporal unrolling is more effective in such systems, as it conserves resources and balances memory access with computation cycles.

Before examining the state-of-the-art works in research and industry that make use of spatial and temporal unrolling it is useful to summarize the main characteristics, pros and cons of these type of design approaches. Most of the reviewed works will incorporate a balance between the two of these.

The Characteristics of Spatial Unrolling:

- **Maximizing Parallelism:** Spatial unrolling is optimal for compute-bound systems, where the primary bottleneck is the availability of compute resources rather than memory latency or bandwidth. Each operation runs in parallel, allowing for significant performance gains by increasing the number of PEs in the design.
- **Latency Reduction:** Spatial unrolling can significantly reduce the latency of a task, as more operations are performed simultaneously.
- **Limitations of Spatial unrolling:**
  - High demand for hardware resources (e.g., area on FPGAs or logic gates on ASICs).
  - Can lead to inefficiencies if the system is memory-bound, as the parallel PEs may become underutilized waiting for data.

The Characteristics of Temporal Unrolling:

- **Resource Conservation:** Temporal unrolling uses fewer hardware resources at a time but requires more cycles to complete a task.
- **Efficiency in Memory-bound Systems:** Since data needs to be fetched and processed over multiple cycles, temporal unrolling is useful when the system is memory-bound—that is, when the latency and bandwidth of memory access are bottlenecks in the design. By reusing the same computational elements over time, temporal unrolling can balance the slower memory access rates.
- **Limitations of Temporal unrolling:**
  – Longer execution time due to sequential processing.

Dataflow Considerations for Accelerator Design: Different dataflow architectures focus on maximizing various types of data reuse, optimizing for specific workloads like convolution, matrix multiplication, or fully connected layers in neural networks. These architectures include:

- **Weight-Stationary Dataflow:** Weights remain in local memory while inputs are streamed through, maximizing weight reuse. This is ideal for convolutional layers where filters are applied repeatedly to different inputs.
- **Output-Stationary Dataflow:** Intermediate results (e.g., partial sums) are accumulated and reused locally, reducing the need for memory writes. This approach is common in matrix multiplication.
- **Row-Stationary Dataflow:** Optimizes the reuse of both input and weight data by focusing on rows of the input matrix and weights. It balances the reuse of multiple types of data.

By designing custom dataflow architectures for specific operations in neural networks, accelerators can optimize for memory bandwidth, minimize energy consumption, and enhance overall performance.

Data reuse is a crucial aspect of efficient accelerator design, particularly for applications like TinyML, where power and memory constraints are significant. By maximizing input reuse, output reuse, and weight and activation reuse, dataflow architectures can deliver higher performance with lower energy consumption. Optimizing dataflow for specific tasks such as convolutions or matrix multiplications ensures that FPGAs, ASICs, and other accelerators can achieve the necessary throughput and efficiency for edge inference and other real-time applications.

Continuing with Fig. 13, the next branching on the Spatial Unrolling branch introduces the idea of granularity. Granularity here refers to the compute capabilities of the processing element. These could be MAC units or they could be simpler logic elements. This is another factor to consider when designing custom implementations. First we discuss fine grained processing elements.

Fine Grained Processing Elements (FPGAs): Field-programmable gate arrays, abbreviated as FPGAs, offer custom hardware design capabilities through a reconfigurable implementation based on routing small fine-grained processing elements together. Fig. 15 shows a stylized architecture of a FPGA, typically there are columns of soft logic blocks, memory blocks (Block RAMs (BRAMs)), and arithmetic blocks (usually Digital Signal Processing (DSP) blocks) that can be connected together. The soft logic blocks typically contain Look-up Tables LUTs as seen through the OpenFPGA soft logic block (see [72] for details). These LUTs are used to model Boolean logic functions; for example, an LUT-6 block will take 6 inputs and map them to one output. Different vendors use different LUT and soft logic processing elements to get a good trade-off between compute elements (i.e. logic density) and routing. These simple logic-based processing elements make FPGAs fine-grained.

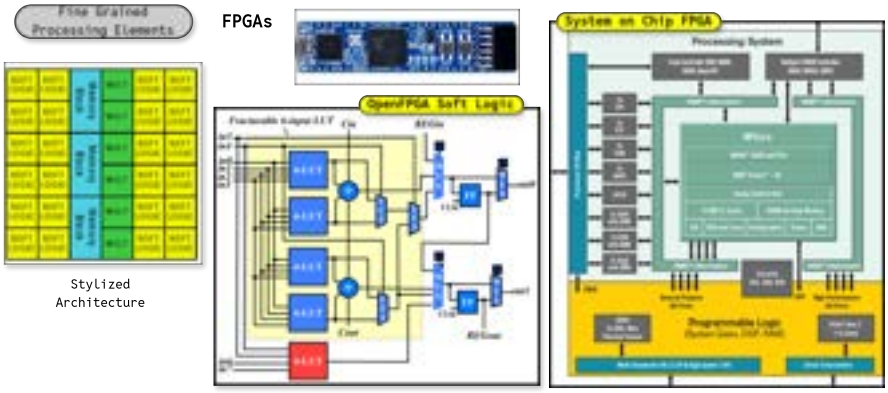


Fig. 15: Fine grained spatial unrolling through FPGAs. The figure shows a picture of a FPGA board available from diligent (CMOD A7), A stylized architecture showing FPGA composition, an expanded view of one possible soft logic block and finally now FPGAs can be developed into System on Chip (SoC) styled implementations.

FPGA Vendors: Xilinx (acquired by AMD) and Intel (through its acquisition of Altera) dominate the commercial FPGA market, providing high-performance devices that are widely used. These companies have continually pushed the boundaries of FPGA architecture by enhancing their devices with high-speed transceivers, large amounts of logic elements, and embedded resources such as DSP slices and memory blocks. In the next section, we will discuss the toolchains that these vendors offer to translate from algorithm to implementation. Recently, open source toolchains for FPGA development have allowed the development of custom FPGA fabrics offering even more customization [73, 72][4]. In recent years the main vendors, along with emerging vendors like Lattice, have started offering smaller form factor, lower power high density FPGAs. Often these are packaged as a System on Chip (SoC),

[4] The next section will discuss more about toolchains for TinyML deployment

these are referred to as embedded FPGAs (eFPGAs). A good example is the Xilinx Zynq7020. In addition, FPGAs like the Lattice iCE40 series are specifically designed for always-on, low-power inference.

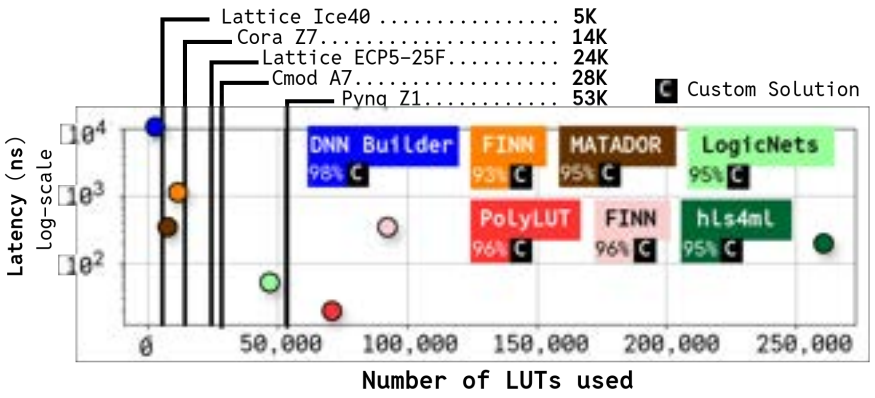


Fig. 16: Figure showing a general trend among FPGA implementations in terms of latency and throughput. The vertical lines indicate the kind of FPGA and eFPGA platforms that are available and their respective LUT availability. A key point to note is that maximally unrolling the model is not so suitable for TinyML - even for a simple application like MNIST that was used for the implementations in this figure. Only three approaches fall into the TinyML territory.

FPGA Design Trends: FPGA implementations that may be regarded as TinyML typically can be divided into two categories; parameterized compute engines that use a combination of spatial and temporal unrolling to allow for greater control in resource usage (FPGA Route 1) [74, 75, 76, 77], or alternatively, custom implementations that maximally unroll the full optimized model into a wired logic solution (FPGA Route 2). These directions have been visualized in Fig. 16. Notice that most of the solutions are well beyond the LUT availability of embedded FPGA platforms seen in the vertical lines. The selected solutions all use an automated pipeline to deploy models on the FPGA. This will be elaborated on further in the next section. For now, the focus is the two types of implementation.

FPGA Route 1 (Parameterized Compute Engine): One of the most popular implementations on this route is FINN from Xilinx [74, 75]. By having design knobs that control the amount of spatial vs. temporal unrolling, it is possible to create a more resource-fugal design most implementations refer to this as the *reuse factor* [78, 77]. In all cases, the implementations use a row-stationary data flow design. This design is essentially a matrix-vector multiplication kernel that can be modified per model. This design thinking is shared by other implementations such as DNN Builder [77], hls4ml [78] and FracBNN [76]. All these works have

favored either binary quantization or 2bit quantization of the weights and activation functions.

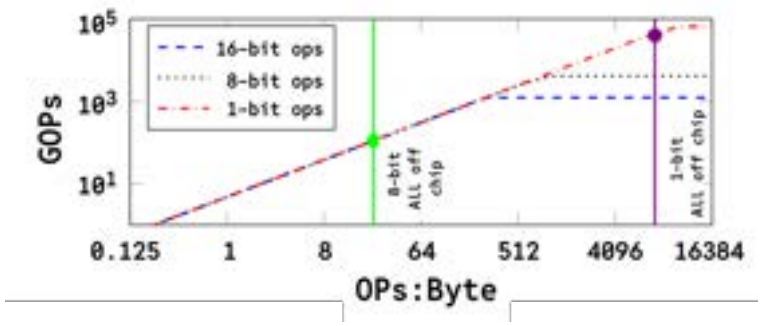


Fig. 17: Figure showing the roofline diagram for the ZU19EG FPGA adapted from [74]. Quantization plays a significant role in whether the implementation is compute or memory bound.

To understand the impact of quantization more quantitatively, Fig.17 shows the roofline diagram for a particular FPGA platform. The x-axis of the graph represents the operations performed per byte of data retrieved from memory. For low quantization, the majority of the data is retrieved off-chip. For single bit quantization, all the data can be kept on chip. Notice the dashed red line; where the line is diagonal, the implementation is memory bound; however, where the line is horizontal, the implementation is compute bound. Therefore, the binary implementations are better for achieving better throughput but return poorer performance from the model - notice how the DNN Builder (12bit quantization) gives much poorer latency but with much lower LUT usage. This is an important trade-off to consider for FPGA implementations.

FPGA Route 2 (Maximal Spatial Unrolling): The alternative to finding the optimal trade-off between spatial and temporal unrolling is to maximally unroll the model. The key is to map the computation efficiently to the fine-grained LUTs. This is the approach taken by PolyLUT [79] and the works it builds from like LogicNets [80] and LUTNets [81]. The approach here is mainly concerned with maximizing throughput over LUT usage. Typically this results in deployment on larger and more power hungry FPGA platforms.

MATADOR [82] explores the maximal unrolling of TMs. The difference is the sparsity that is seen in the TMs allows for both fast throughput but also more frugal LUT usage meaning the implementations can fit on resource constrained eFPGAs. These maximally unrolled solutions are custom for each model. Therefore it is important that they come with an automation flow that can easily convert the co-designed model into implementation. This is explored further in the following section focusing on automation toolchains.

Summarizing FPGAs for TinyML: FPGAs offer a highly versatile and efficient platform for accelerating machine learning workloads, particularly for TinyML and edge computing applications. With innovations in spatial unrolling and systolic array designs, FPGAs can achieve low-latency and high-throughput inference, even in power-constrained environments [83]. However, the key is to use parameterized compute engines like FINN and FracBNN or implementations that support extreme sparsity MATADOR. As a result, both commercial FPGAs (e.g., Xilinx, Lattice, and Intel) and open-source solutions are becoming critical tools in the development of custom accelerator implementations for TinyML.

Coarse Grained Processing Elements: The multiply accumulate operations for Neural Network architectures can be implemented into more coarse grained processing elements where each element will be a MAC unit. ASIC implementations allow for greater flexibility in the size and functionality of these units.

The Google Edge TPU (Tensor Processing Unit) is a hardware accelerator designed to enhance machine learning inference at the edge from Google. It is incorporated into Google's Coral platform, which aims to enable efficient AI applications on resource-constrained devices. The Edge TPU is optimized for running TensorFlow Lite models, making it particularly suitable for TinyML applications where low power consumption and high performance are critical. TensorFlow Lite models enable developers to leverage pre-trained models for various applications - this will be covered more in the following section.

Edge TPU provides a performance throughput of up to 4 trillion operations per second (TOPS) while maintaining low power consumption. Its architecture is designed to efficiently execute neural network operations, particularly those commonly used in deep learning models. The Edge TPU employs a systolic array architecture, which allows for efficient data flow and parallel processing of matrix operations. This design is particularly beneficial for convolutional neural networks (CNNs), which rely heavily on matrix multiplications [84]. It features a significant amount of on-chip memory, which reduces the need for external memory access and minimizes latency by processing data locally, reducing the need for cloud-based inference. This is crucial for real-time applications requiring quick responses [85]. The Edge TPU is designed to operate within a power envelope suitable for battery-powered devices, making it ideal for IoT applications. Its efficiency allows it to perform complex computations without draining the device's battery, typically around 2 watts, which is essential for deployment in resource-constrained environments [86].

The main challenge is model size limitation, which is when the model size does not meet the limits of its on-chip memory. Additionally, although the Edge TPU primarily supports TensorFlow Lite, developers using other frameworks may need to convert their models to ensure compatibility, which can introduce additional complexity [87].

Apple Neural Engine (ANE) is a dedicated neural processing unit (NPU) integrated into Apple's custom silicon, such as A-series and M-series chips. It is designed to accelerate machine learning tasks on iOS and macOS devices. ANE has developed significantly through its generations, with each providing considerable improvements in throughput and efficiency for machine learning tasks, from basic facial recognition in the A11 Bionic to advanced real-time processing in the A18 chip as listed in Table 2. ANE has become a critical component in enabling intelligent applications across Apple's ecosystem. As the demand for on-device AI continues to grow, the ANE plays a substantial role in enhancing its capabilities like privacy-preserving by enabling on-device processing of models without needing cloud computation; tightly coupled with the CPU GPU on Apple SoCs (System on Chips) facilitates sharing of resources. Furthermore, ANE supports the bfloat16 (brain floating point 16-bit) format, making it useful for training large models where memory efficiency is critical. The multi-core nature of the ANE accelerates complex neural networks by parallelizing operations like convolutions, matrix multiplications, and activation functions, thereby offering beneficial features for TinyML applications and edge computing, as well as mobile and wearable technologies. Various Applications leveraging the ANE from Real-time Image Processing (e.g., object detection, facial recognition), Natural Language Processing on voice-enabled devices (Siri), and Health Monitoring through real-time sensor analysis in devices like Apple Watch and Augmented Reality (AR) applications. ANE works seamlessly with Apple's Core ML framework, making it easy for developers to deploy machine learning models on Apple devices. The main limitation is that unlike traditional GPUs or CPUs, which can be programmed for a wide range of tasks, the ANE is highly specialized for neural networks. For applications outside ML, it does not provide as much flexibility as a general-purpose processor. ANE is only available in Apple's ecosystem, limiting its accessibility to developers who work with non-Apple hardware. Core ML is Apple's machine learning framework that allows developers to integrate pre-trained models into their apps. The framework decides whether to run the model on the CPU, GPU, or ANE based on the model's structure and the available hardware. ANE accelerates specific operations such as CNNs, and RNNs. Matrix Multiplications are common in fully connected networks. Core ML can automatically optimize models for further improving performance by quantizing models into low-precision formats, which the ANE can process faster.

Qualcomm's Hexagon DSP (Digital Signal Processor) is a specialized hardware accelerator designed to optimize performance for various computational tasks, particularly in mobile, embedded systems, and IOT. It specifically focusing on real-time, low-power AI tasks, making it ideal for applications in tinyML (tiny Machine Learning) where resource constraints are a significant concern [90, 91]. The Hexagon DSP architecture is integrated into Qualcomm's Snapdragon SoCs. The architecture includes features like vector processing units (VPUs), supported by Hexagon Vector Extensions (HVX), to enable more efficient vector processing capabilities. These VPUs are designed to handle multiple data points simultaneously, leveraging the principles of Single Instruction Multiple Data (SIMD) architecture. This capability allows the Hexagon DSP to execute operations on large data sets efficiently, which

Table 2: Development of Apple Neural Engine used in iPhones [88, 89].

Chipset	Peak TOPS	Technology (nm)	Core Count	Other Features	Used In
A18 Pro	35	Enhanced 3 (N3E)	16	20% faster and 2x faster ray tracing than A17	iPhone 16 Pro, 16 Pro Max
A17 Pro	35	3 (N3B)	16	20% faster and 4x faster ray tracing than A16	iPhone 15 Pro, 15 Pro Max
A16 Bionic	17	4 (N4P)	16	CPUs with 20% less power, 50% more GPU memory bandwidth than A15	iPhone 14 Pro, 14 Pro Max, 15, 15 Plus
A15 Bionic	15.8	5	16	40% faster CPU A12	iPhone 13, 13 Pro, 14, 14 Plus
A14 Bionic	11	5	16	40% faster CPU, 30% faster GPU compared to A12	iPhone 12 series, iPad Air (4th gen)
A13 Bionic	6	7	8	20% faster CPU and 15% lower power	iPhone 11 series, iPhone SE (2nd gen)
A12 Bionic	5	7	8	9x faster, 90% lower power consumption	iPhone XS, XR, iPad Pro (3rd gen)
A11 Bionic	0.6	10	2	First Neural Engine, Real-time ML tasks, like Animoji and Face ID	iPhone 8, iPhone X

Note: N3E, N3B, and N4P refer to TSMC's advanced semiconductor technology nodes. N3E is an enhanced 3nm node, N3B is the baseline 3nm node, and N4P is an optimized 4nm node, offering different trade-offs in power efficiency, performance, and cost.

is particularly beneficial for applications requiring real-time processing, such as audio, video processing, and sensor data, as well as machine learning inference tasks [91, 92, 93]. Additionally, it supports a range of data types and operations, including fixed-point arithmetic, which is often used in machine learning models to reduce computational load and power usage [92]. The Hexagon DSP's architecture allows for the efficient execution of neural networks and other machine learning algorithms, enabling on-device inference without the need for cloud processing.

5.3 Application-Specific Integrated Circuit (ASIC)s:

offer the most freedom when developing coarse-grained processing elements. Designers have control over the precision of the computation and its hardware composition. As shown in Fig. 12, the ASIC has the smallest power consumption of the platform types, but can offer competitive GOP/s for some implementations.

Generally speaking, the advantages of ASICs mentioned above must be considered against their high cost (most expensive route compared with CPUs/GPUs/FPGAs) and longest time to market. Shuvo et al offer a summary of current ASIC implementations for Neural Networks [94]. Their review indicates that ASIC designs are focused on being able to service many different NN structures, most commonly focusing on VGG-16, AlexNet. Almost all of them focus on image classification,

segmentation, or analysis. One of the key design aspects here is the different modalities; designs now offer variable precision MAC units and as such variable GOP/W. They also have different architectural modules that may be switched off depending on the application.

5.4 Compute-in-Memory (CiM):

CiM is gaining traction as an effective approach for accelerating TinyML applications by addressing the memory wall challenge inherent in traditional computing systems, called Von Neumann. CiM offers system enhancements by reducing the energy and inference latency required for data movement between memory and processing units. These environments are promising solutions for systems typically operating on resource-constrained devices such as TinyML. CiM architectures mitigate this bottleneck by performing computation directly within the memory arrays; the in-situ data processing allows the running of complex machine-learning tasks on devices that operate on battery power [95, 96]. CiM further reduces the latency issue by allowing for parallel data processing directly within memory cells [97]. Low latency and parallelism are sectionals, especially in applications requiring real-time analysis. In addition, IMC reduces the exposure of TinyML applications to data corruption and noise, which are common challenges in resource-constrained environments as the computations are directly within memory [98, 99]. CiM can be implemented using digital or analog approaches, each leveraging different memory technologies as described in the following sections.

5.5 Digital in-memory computing (DiMC):

DiMC relies on binary arithmetic and logic operations, such as AND, OR, and NOR, to perform basic computational tasks directly within memory arrays. The DiMC architectures typically utilize standard memory technologies based on charge storage, such as static random access memory (SRAM), dynamic random access memory (DRAM), and Flash memory as storage mediums. The memory architecture in DIMC generally follows a row/column organization, where memory cells perform computations in parallel across rows, resulting in energy-efficient bulk operations like matrix-vector multiplication (MVM). The binary matrices and vectors are stored in memory, and MAC operations are conducted using the memory's inherent properties. In SRAM arrays capacitors can be added to SRAM arrays to accumulate MVM charge, enabling efficient matrix computations. Similar operations can be performed in DRAM by redistributing charge across multiple rows or columns.

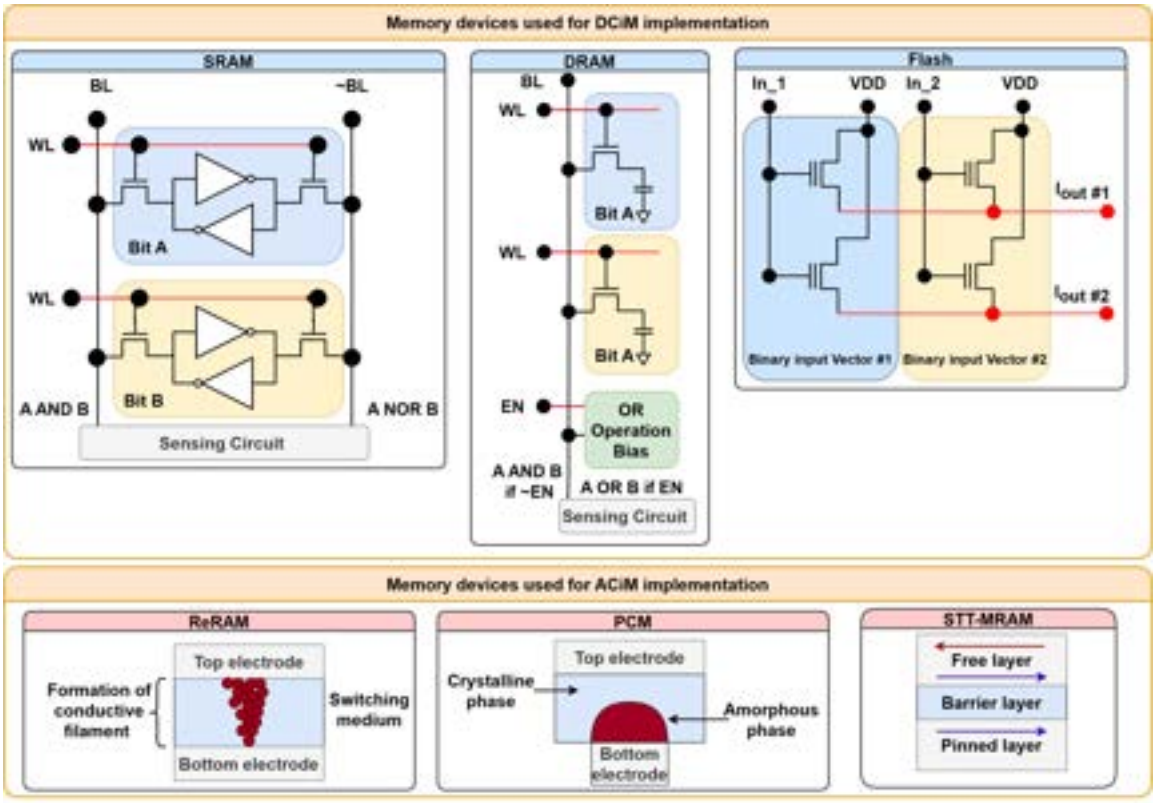


Fig. 18: CMOS-based Digital Memory Devices and Emerging Analog Memory Devices Used in Compute-in-Memory Architectures for Machine Learning Applications.

5.5.1 Static Random Access Memory:

SRAM plays a pivotal role in DiMC due to its fast access times and high cycling endurance. Each SRAM cell is typically composed of six transistors (6T) that form a bistable latch, enabling rapid switching between binary states. With sub-nanosecond access times (< 1 ns) and high endurance ($> 10^{16}$ cycles), SRAM is well-suited for DiMC applications that demand frequent data reads and writes, such as real-time AI inference and deep learning accelerators [100, 101].

One of the critical benefits of SRAM in DiMC is its ability to perform MVM and logical operations directly within the memory, as shown in Fig. 18, significantly reducing data movement and thus improving energy efficiency. However, due to the need for continuous power to maintain data, SRAM's high power consumption is a notable drawback. Additionally, SRAM's large cell size limits its scalability

compared to other memory types [101]. Despite these challenges, SRAM remains a critical component in DiMC systems where low latency and high computational throughput are priorities.

5.5.2 Dynamic Random Access Memory:

DRAM is another important memory type used in DiMC, primarily due to its high density and low cost per bit. DRAM cells consist of a capacitor and a transistor, enabling a smaller cell size around $6F^2$, which allows for higher storage density compared to SRAM [100, 102]. In DiMC, DRAM can execute basic logic operations such as AND and OR by manipulating charge redistribution across capacitors, as shown in Fig. 18. Although DRAM's access times are longer than SRAM's (around 10 ns), its capacity for large-scale data storage makes it suitable for applications involving extensive data manipulation [100, 101]. A significant advantage of DRAM in DiMC is its ability to handle bulk bitwise operations, enhancing parallel computation. However, DRAM's need for periodic refreshing to maintain data integrity presents a challenge in terms of energy efficiency. Moreover, its lower endurance compared to SRAM, typically around 10^9 cycles, further limits its applicability in scenarios requiring frequent data updates [100, 101]. Despite these challenges, DRAM's cost-effectiveness and scalability make it a valuable component in DiMC architectures for data-intensive applications.

5.5.3 Flash Memory:

Flash memory, particularly in NOR and NAND configurations, is widely used in DiMC due to its non-volatility, allowing data retention without a continuous power supply. Flash memory cells are smaller, enabling a cell size of around $4F^2$ to $10F^2$, which contributes to high storage density [100, 101]. In DiMC, Flash memory can perform MVM by modulating gate voltages and storing matrix elements as charges on floating gates, as shown in Fig. 18, with typical read times of less than 100 ns and write latencies exceeding $\sim 10\mu$ s [100, 101]. NOR Flash is favored for DiMC due to its ability to support parallel access, which is critical for high-throughput operations. One of Flash's key advantages is its scalability, making it ideal for large-scale storage applications. However, Flash memory has lower endurance, typically around 10^5 cycles, and higher write energy requirements, often requiring voltages exceeding 10 V [100, 101]. Despite these limitations, the non-volatility and high density of Flash memory make it an attractive choice for DiMC applications that require long-term data storage and infrequent writes, such as machine learning inference and data archiving.

In summery DiMC is advantageous for applications requiring high accuracy and reliability, as digital circuits are less susceptible to noise and variability than their analog counterparts. Furthermore, DIMC can be integrated with existing digital processing architectures, making it easier to adopt in current systems.

5.5.4 Case Study of DiMC: XNOR-SRAM

The XNOR-SRAM is a 65-nm CMOS ML accelerator based on the DiMC concept [103], as shown in Fig. 19a. XNOR-SRAM offers efficient integration of computation within the memory array to enhance performance in binary and ternary DNNs. This architecture embeds ternary-XNOR-and-accumulate (XAC) operations directly into a 256-by-64 SRAM bit-cell memory array. By enabling simultaneous activation of all rows, XNOR-SRAM achieves analog accumulation of bitwise ternary-XNOR outputs on the read bit-line (RBL), which are then digitized using an 11-level flash ADC. Operating at a supply voltage of 0.6V, this direct in-memory computation eliminates the need for row-by-row memory access, reducing energy consumption and computation latency in conventional digital implementations.

Fig. 19b shows that the XNOR-SRAM achieves a 33× improvement in energy efficiency and a 300× better energy-delay product while performing ternary-XAC operations with a 403 TOPS/W performance. This efficiency is accompanied by high test accuracy, reaching 88.8% for the CIFAR-10 dataset and 98.8% for MNIST when using ternary precision, as detailed in Fig 19c. These results demonstrate the balance between energy efficiency and computational accuracy in this DiMC architecture.

The architecture supports binary weights and activations (+1, −1) and ternary inputs (+1, 0, −1), making it highly versatile for DNN applications. The linear quantization scheme incorporated into the flash ADC enhances CIFAR-10 accuracy by optimizing voltage intervals, improving the result from 85.7% to 88.8%. This improvement reflects the robustness of the XNOR-SRAM design in preserving precision despite the challenges of analog processing.

5.6 Analog in-memory computing (AiMC):

AiMC extends the conventional digital computing paradigm by capitalizing on continuous, non-binary resistive, or charge states in memory devices to perform complex arithmetic operations, such as MVM and finite-state machine (FSM) processing. Unlike digital approaches that rely on sequential processing, AiMC executes these operations directly within the memory array, enabling highly parallel and energy-efficient computation. The core principle of AiMC lies in leveraging Ohm's law and Kirchhoff's Law to perform in-situ arithmetic. In this approach, trained model parameters (such as weights or states) are encoded as resistive values within the memory array, while input data (such as encoded features) is applied as voltage signals. The interaction between these two operands generates accumulative currents at the array's output nodes, representing the computed results of MVM or other tasks. Resistive or memristive memory technologies, including Resistive RAM (ReRAM), Phase Change Memory (PCM), and Magnetoresistive RAM (MRAM), are essential to AiMC as these cells represent the learning elements. These technologies provide non-volatile storage, high density, and analog programmability, making them ideal for encoding model parameters and executing computations in memory. The analog

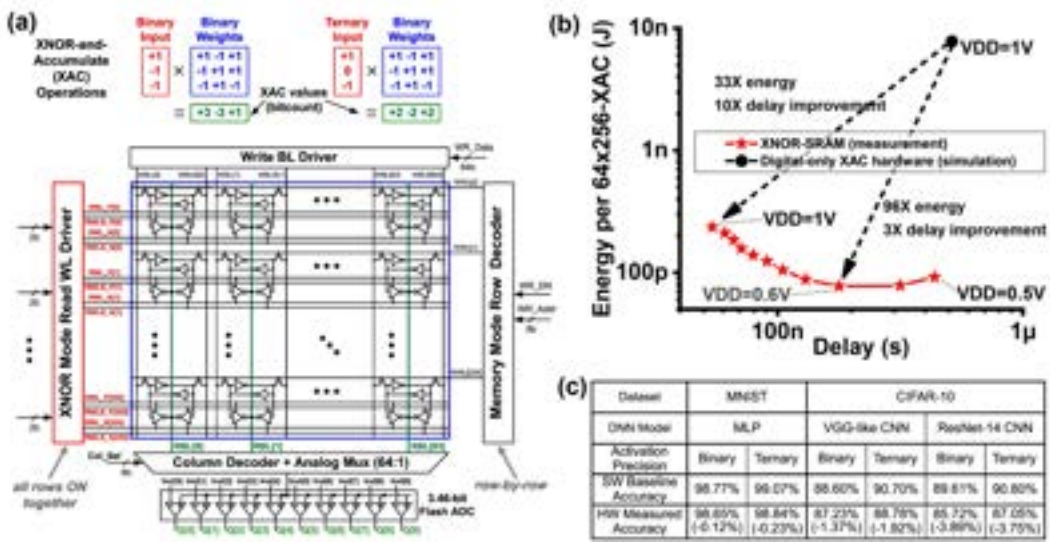


Fig. 19: XNOR-SRAM Overview: (a) Architecture: A 256-by-64 SRAM array performing in-memory ternary-XNOR-and-accumulate (XAC) operations with analog accumulation and flash ADC digitization. (b) Energy and Delay: XNOR-SRAM achieves 33× higher energy efficiency and 300× better energy-delay product than a digital baseline, with 403 TOPS/W at 0.6V. (c) Accuracy: Achieves 98.8% on MNIST and 88.8% on CIFAR-10 using ternary precision, combining efficiency and precision.

outputs from AiMC can either be processed further in the analog domain or converted to the digital domain for subsequent processing. For conversion, techniques such as Analog-to-Digital Converters (ADCs), Current Sense Amplifiers (CSAs), or Transimpedance Amplifiers (TIAs) are employed.

AiMC has advantages over DiMC, particularly for machine learning applications and workloads that demand high throughput through massively parallel operations within memory arrays and low power consumption. Moreover, it leverages the physical properties of memory devices for computation, allowing higher computational density and faster execution of tasks like MVM. By integrating storage and computation, AiMC offers a promising solution to overcome the memory-wall bottleneck in traditional computing. In the following sections, we delve deeper into the operational principles and unique advantages of ReRAM, PCM, and MRAM in AiMC applications.

5.6.1 ReRAM Memory:

ReRAM is a non-volatile memory technology that has emerged as a key enabler for AiMC due to its inherent resistive switching properties. ReRAM operates by modulating the resistance of a memory cell between a Low Resistance State (LRS) and a High Resistance State (HRS) using electrical SET and RESET pulses. Additionally, certain ReRAM devices can stabilize intermediate resistance states, enabling them to store analog data, an essential feature for performing in-memory computations like MVM [104].

Structurally, ReRAM consists of a simple yet versatile metal-insulator-metal (MIM) stack, where the insulator layer, typically made of transition metal oxides like hafnium oxide (HfO_2) [105] or titanium dioxide (TiO_2) [106], governs the resistive switching behavior. The switching mechanism relies on the formation and dissolution of conductive filaments (CFs) within the insulator, which are driven by the migration of oxygen vacancies or metallic ions. During the SET process, CFs bridge the electrodes, creating an LRS, as shown in Fig. 18. Conversely, the RESET process disrupts these pathways, transitioning the cell to the HRS. This robust switching mechanism enables ReRAM cells to act as tunable resistors, capable of encoding both binary weights by switching between the LRS and HRS and analog weights by leveraging the ability to stabilize resistance levels within the intermediate states between LRS and HRS [107].

For AiMC applications, ReRAM arrays are typically organized in dense two-dimensional crossbar architectures. This configuration enhances computational throughput by facilitating parallel operations across the ReRAM array. Each ReRAM cell in the crossbar represents a synaptic weight, and by applying input voltages (V_i) to the rows of the array, the cumulative current (I_i) measured at the columns represents the computational result. This process directly leverages physical laws such as Ohm's Law and Kirchhoff's Current Law to perform MVM efficiently:MVM = $\sum_{i=1}^{m}(G_i V_i)$ where G_i is the conductance of the ReRAM cell [108]. This in-memory approach eliminates the overhead associated with data movement between memory and processing units, offering significant energy and speed advantages. ReRAM's non-volatile feature eliminates the need for refresh cycles, significantly reducing power. Its ability to stack multiple layers of crossbar arrays further enhances storage density and computational capacity, making it well-suited for AI workloads and neuromorphic computing [108, 109]. Despite its promising features, ReRAM faces challenges such as device variability, limited write endurance, and non-linear switching characteristics [110, 111]. Addressing these challenges will require advancements in materials engineering, device design, and fabrication techniques.

5.6.2 PCM Memory:

PCM is another promising candidate for AiMC, leveraging the unique properties of phase change materials such as $Ge_2Sb_2Te_5$ to achieve non-volatile storage. PCM exploits Joule heating to induce rapid transitions between amorphous (HRS) and

crystalline (LRS) states, as shown in Fig. 18. By carefully tuning the energy of these pulses, intermediate resistance states can be achieved, allowing for precise analog storage. This characteristic is particularly beneficial for AiMC, where the continuous resistance range enables accurate representation of model parameters [112, 113].

PCM cells often feature a mushroom-shaped architecture, with a bottom electrode designed to confine heat and current for energy-efficient operations. In AiMC applications, PCM can be arranged in dense two-dimensional crossbar architectures, similar to ReRAM. PCM arrays are used to execute MVM directly in memory by encoding weights as conductance values and applying input voltages across rows of the array. The resulting cumulative currents at the output nodes correspond to the weighted sums, calculated using Ohm's and Kirchhoff's laws. PCM's accumulative behavior further supports iterative adjustments during tasks like neural network training, where conductance can be progressively tuned [114, 114]. However, challenges such as high programming energy, thermal crosstalk, and material fatigue due to repeated phase transitions must be addressed. Despite these limitations, PCM remains a powerful tool for AiMC due to its precision, stability, and scalability [115].

5.6.3 Spin-Transfer Torque Magnetic RAM:

STT-MRAM is an advanced non-volatile memory technology that leverages spintronics to provide high-speed, low-power, and high-density memory solutions. STT-MRAM operates by manipulating the spin of electrons to switch the magnetic state of a Magnetic Tunnel Junction (MTJ), which serves as the core building block of this memory technology. The MTJ consists of two ferromagnetic layers separated by a thin insulating barrier. One layer is fixed (the reference layer), while the other (the free layer) can change its magnetization direction, as shown in Fig. 18. When a current is passed through the MTJ, the spin-polarized electrons exert a torque on the magnetic moments of the free layer, enabling it to switch between parallel (LRS) and anti-parallel (HRS) configurations. This mechanism allows for binary data storage [116].

For AiMC applications, STT-MRAM can be arranged in dense two-dimensional crossbar architectures, similar to other in-memory computing technologies such as ReRAM. This structure allows for efficient MVM operations, where each cell represents a synaptic weight in a neural network. The rows and columns of the array can be activated simultaneously to perform computations, enabling efficient and rapid processing of large data volumes in learning applications. The resulting cumulative currents at the output nodes correspond to the computed results, leveraging Ohm's Law for direct in-memory processing [117].

STT-MRAM offers several advantages. Firstly, its non-volatility allows it to retain data even when power is removed. Secondly, STT-MRAM provides high-speed performance, with write speeds in the nanosecond range, faster than traditional Flash memory and comparable to SRAM. Additionally, STT-MRAM offers high density, achieving high integration densities that allow more data to be stored in a

smaller physical area. Finally, it has high endurance, being capable of withstanding many write cycles without degradation [118, 119, 120].

However, STT-MRAM also faces several challenges that need to be addressed. One such challenge is write energy, as the energy consumption during write operations can be higher than that of SRAM, which may pose challenges in energy-sensitive applications. Another issue is process variations, as STT-MRAM devices are sensitive to variations in manufacturing processes, which can lead to inconsistencies in performance and reliability [118]. Thermal stability is also a concern, as elevated temperatures can affect the reliability and performance of STT-MRAM cells [121, 122]. Finally, the complexity of control circuits is another challenge, as precise control of the write and read operations in STT-MRAM can complicate the design of the associated circuitry, potentially increasing the overall system complexity [123, 124].

5.6.4 Case Study of AiMC: The IMBUE

The IMBUE in [125] introduces a novel accelerator design aimed at improving the energy efficiency and performance of Tsetlin Machines using in-memory computing principles as shown in Fig. 20. The IMBUE architecture leverages ReRAM technology, integrating computation and memory to address von Neumann bottlenecks in traditional machine learning ML systems. The IMBUE accelerator interacts directly with the input Boolean feature as digital voltages by translating them into current paths and exploiting the simplicity of Tsetlin Automata for logic inference tasks. This Boolean-to-Current mechanism enabled the streamlining of the inference process and eliminated the requirement for high precision analog to digital converters, thus significantly reducing circuit overheads [97, 126]. The design is organized into partial clauses to handle the inherent non-linearity in ReRAM devices and to mitigate sneak currents. IMBUE minimizes power consumption by exploiting the sparse activation of TAs (mostly ”exclude” actions), leading to a remarkable up to 12.99x improvement over state-of-the-art binarized convolutional neural networks and up to 5.28x improvement over digital TM implementations. Critical results include energy efficiency evaluations across datasets like MNIST and Fashion-MNIST, demonstrating up to 5.283x energy reduction. By addressing the challenges associated with traditional computing architectures and leveraging the unique properties of in-memory computing, It is expected to see increasingly sophisticated implementations that push the boundaries of what is possible in TinyML and artificial intelligence.

5.7 Logic-Based ASICs: Tsetlin Machine Accelerators

The design in [127] presented a custom ASIC accelerator to achieve energy-efficient machine learning tailored for IoT applications. The ASIC utilizes the Tsetlin Ma-

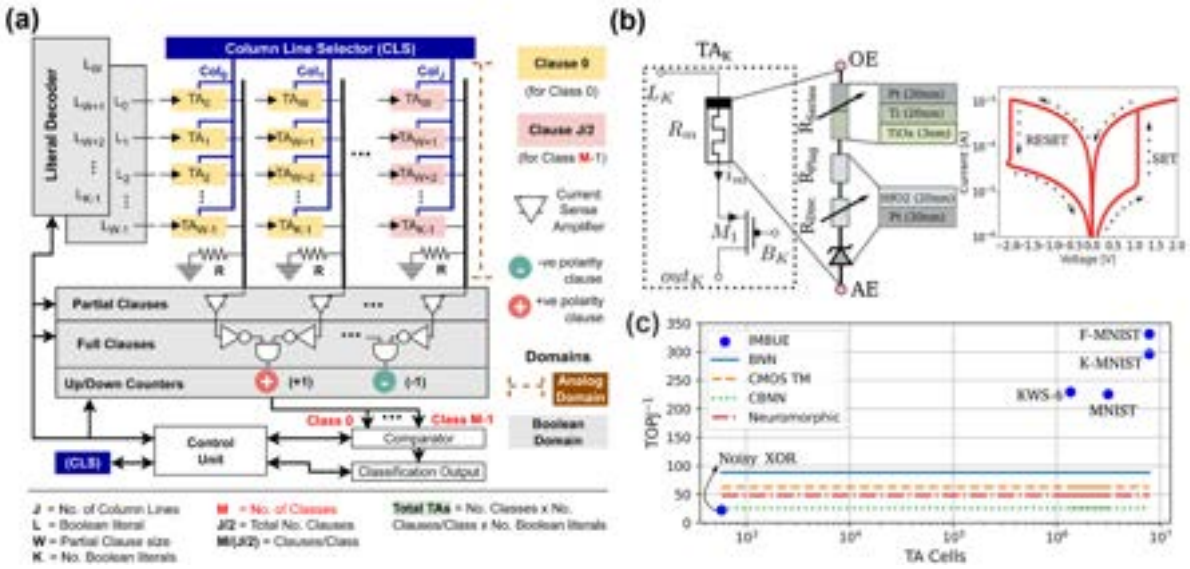


Fig. 20: (a) TM inference architecture using IMBUE.(b)An equivalent circuit for a TA is represented by a 1T1R cell, and (c) IMBUE energy efficiency for TM inference is applied to various datasets.

chine, a logic-based algorithm that replaces computationally intensive neural network operations with propositional logic, significantly reducing energy consumption and hardware complexity. The binarized data encoding process reduces the feature space while maintaining classification accuracy. For example, feature encoding was reduced by 2.67x in the binary Iris dataset, enabling a compact and efficient hardware implementation. The architecture further supports on-chip learning, which allows real-time adaptability to changing environments, and uses an FPGA-based platform for rapid hyperparameter tuning.

The ASIC performance on the Iris dataset is a major highlight. Using the TM with the configuration of 16 inputs, 100 clauses, and 3 classes, the design achieved an accuracy of 96.3% for training and 97.0% for testing, comparable to state-of-the-art solutions. The implementation demonstrated an area of 0.386 mm^2, operating at 33.3 MHz, and a training power of 1.85 mW, as shown in Table 3. Energy consumption per datapoint was measured at 55.6 pJ for training and 30.6 pJ for inference, underscoring its suitability for low-power IoT environments. The ASIC leverages 65 nm technology, operating at a voltage of 1.0 V, with an architecture that prioritizes efficiency through logic-based clauses and power-gated reinforcement blocks, achieving performance of 62.7 Top/J, surpassing other accelerators such as BNNs. By integrating energy-efficient hardware with adaptable logic-based learning, this ASIC sets a new prototype for edge AI in IoT applications.

Table 3: Comparison of ASIC and FPGA implementations for noisy XOR and binary Iris datasets [127].

Feature	Noisy XOR (ASIC)	Noisy XOR (FPGA)	Binary Iris (ASIC)
# Inputs, Clauses, Classes	12, 10, 1	12, 10, 1	16, 20, 3
ASIC Area (mm^2)	0.246	—	0.386
Frequency (MHz)	118	110	33.3
Training Time (ms)	4.24	4.55	1.80
Avg. Power (Training)	16.8 µW	1.10 W	1.85 mW
Energy/Datapoint (Training)	142 fJ	10.0 nJ	55.6 pJ
Energy/Datapoint (Inference)	—	—	30.6 pJ

This section examined **ASIC** advancements for efficient computing, highlighting Compute-in-Memory approaches such as **DiMC** and **AiMC**, which utilize technologies like SRAM, DRAM, FLASH, ReRAM, STT-MRAM, and PCM for energy-efficient processing. Case studies, including XNOR-SRAM, IMBUE, and logic-based designs, illustrated the practical benefits of these accelerators in optimizing ML performance and power usage.

5.8 The Generic Route: CPUs and GPUs

5.8.1 RISC-V Accelerators:

RISC-V (Reduced Instruction Set Computing) is an open-source instruction set architecture (ISA) that has gained significant traction in the TinyML community and various computing domains [128]. One of the most compelling features of RISC-V is its modularity. The architecture consists of a base instruction set that can be extended with optional features tailored to specific applications. This design flexibility and The open nature of RISC-V encourage the developers to create customized processors that meet the unique requirements for low-power embedded systems such as CNN accelerators, edge intelligence applications, and network packet processing. The following are the key features of RISC-V:

- **Modularity and Extensions** RISC-V is built around a core integer instruction set that can be enhanced with optional extensions. These modules add specialized functionalities for different use cases. For instance, M (Multiplication/Division) adds support for integer multiplication and division. A (Atomic) enables atomic operations for multi-threaded environments.F/D/Q (Floating Point) supports single, double, and quad-precision floating-point arithmetic. C (Compressed) reduces instruction size to save memory and bandwidth. V (Vector) provides vector-based computation for high-performance and AI workloads.

- **Base ISA Flexibility** RISC-V offers three major base ISAs are RV32I (32-bit), RV64I (64-bit), and RV128I (128-bit) these frozen standards ensure long-term stability and backward compatibility.
- **Flexible Register Design** RISC-V utilizes 32 general-purpose registers (x0–x31). The zero register (x0) is hardwired to 0, simplifying certain operations. the remaining registers can be customized based on Application Binary Interface (ABI) requirements.
- **Instruction Formats** RISC-V uses six main instruction formats (R, I, S, U, B, and J), all of which are 32-bit wide. This consistency simplifies instruction decoding and improves pipeline efficiency, making RISC-V easier to work with than ARM or x86 architectures.

RISC-V has been implemented in various cores, processors, and SoC platforms, demonstrating its versatility across numerous applications. The PULP(Parallel Ultra-Low-Power Processing) platform is a silicon-proven framework that provides highly energy-efficient solutions for IoT and embedded applications [129]. Developed by ETH Zurich and the University of Bologna, PULP leverages clusters of RISC-V cores with shared tightly coupled data memory. It offers scalability for a wide range of applications, from simple microcontroller systems to advanced multi-cluster accelerators. PULP's architecture is implemented using RISC-V-compatible cores written in SystemVerilog, and it integrates a variety of peripherals, custom accelerators, and interconnect solutions. PULP features several processors, including CV32E40P (RI5CY), a 32-bit, 4-stage core with DSP-specific extensions; Ibex (Zero-riscy), a 2-bit, 2-stage core optimized for control applications and area efficiency; Micro-risky provides a 32-bit, 2-stage core with 16 registers, implementing RV32-EC without a hardware multiplier; CVA6 (Ariane), a 64-bit,6-stage core supporting Unix-like operating systems offering three privilege levels (M, S, U) for effective OS management; and Another notable core is Snitch, a single-stage, 32-bit, 1 stage core optimized for energy efficiency. By externalizing the FPU and focusing on simple, single-issue operations, Snitch aims to maximize computational throughput while keeping power consumption low. The simpler PULP-based microcontroller systems, PULPino and PULPissimo. PULPino is a minimal single-core RISC-V SoC that has attracted attention for its simplicity and effectiveness in embedded applications. PULPissimo, a more advanced version, introduces a logarithmic interconnect that allows multiple access ports between the core and the memory subsystem. It is integrated with a micro-DMA (µDMA), enabling peripherals to interconnect to the memory [129].

5.8.2 Single Board Computer (SBC):

SBC is a compact computing device that integrates all the essential components of a computer onto a single circuit board. SBC has a CPU, memory RAM, and various I/O interfaces for peripheral connectivity. The compact nature of SBCs and the advancements in technology enable them to address increasingly complex computations, including machine learning tasks particularly suitable for at the edge, where limited

resources (capacity and power) and real-time efficiency are critical [130, 131].

Raspberry Pi: is among the most popular SBCs is the Raspberry Pi, developed by the UK-based Raspberry Pi Foundation, established in 2009 with the primary mission of advancing computer science education in schools [132]. The first iteration, the Raspberry Pi Model B, debuted in February 2012, featuring 256MB of RAM and a single-core ARM11 processor at 700 MHz. This initial model was widely utilized in academic and hobbyist projects due to their versatility, affordability,and a rapidly expanding support community [133]. Since then, Raspberry Pi has experienced several iterations, each improving in terms of performance, connectivity, and capabilities. The Raspberry Pi 2 2015 upgraded with a more powerful quad-core ARM Cortex-A7 processor and 1GB of RAM, allowed for more challenging applications [134]. To enhance usability in IoT applications, built-in Wi-Fi and Bluetooth were added to the Raspberry Pi 3, followed in 2016 [132]. The Raspberry Pi 4, released in 2019, marked a significant advancement, with enhanced graphics capabilities, support for dual 4K HDMI outputs, USB 3.0, and a memory capacity of up to 4GB of RAM, establishing it as a practical SBC option for lightweight computing applications [134]. The most recent model, the Raspberry Pi 5, was released in October 2023. It was equipped with a quad-core ARM CPU of 2.4 GHz speed and 8 GB RAM. The Raspberry Pi 5's compatibility with running the in-house Raspberry Pi OS as well as Android 15 makes it well-suited for demanding TinyML applications.

Raspberry Pi is A Pioneering SBC in TinyML: The integration of Raspberry Pi in TinyML applications presents several advantages that enhance the efficiency and effectiveness of deploying tinuML models on edge devices.

- **The versatility and Low cost** One of the primary benefits of using Raspberry Pi in TinyML is its affordability and accessibility. The low cost of Raspberry Pi devices allows for widespread adoption in various TinML applications due to its ability to interface with various sensors, and actuators, which enhance the Raspberry Pi's versatility. The Raspberry Pi's GPIO (General Purpose Input/Output) pins facilitate multiple sensor connections, allowing concurrent data processing, a vital feature for real-time applications that gather data from diverse environmental sources [135]. This local processing reduces latency and minimizes bandwidth usage associated with cloud computing, making it especially valuable for TinyML applications in remote or power-limited environments [136].
- **Small form factor** Another significant advantage of utilizing Raspberry Pi in TinyML is its processing power relative to its small form factor. It supports the efficient execution of lightweight machine-learning models in space-constrained environments [137]. For instance, in a study focused on air quality prediction, the Raspberry Pi was employed for real-time inference to run CNNs for accurate hourly forecasts, demonstrating its ability to handle complex computations at the edge [135]. Moreover, Creating clusters of SBCs further enhances their computational capabilities, allowing for distributed processing and improved performance in more complex ML tasks [138].

- **libraries** Because of the Raspberry Pi's flexibility, developers may take advantage of pre-existing TinyML frameworks and tools, which makes model deployment and prototyping easier[o25]. It has broad support for a variety of IoT and TinyML applications.The main operating system, Raspberry Pi OS is equipped with pre-installed libraries and programming tools, making it the perfect environment for developers working with Python, C++, and Java. furthermore, a vast array of libraries and frameworks that support development across all areas enhance the Raspberry Pi ecosystem, including OpenCV [139], TensorFlow Lite [140], and the Node-RED platform [141]. Additionally, The Raspberry Pi community plays a crucial role in supporting developers through forums, tutorials, and documentation such as GitHub and Stack Overflow

The Raspberry Pi's processing capability allows it to implement lightweight RNNs or LSTM networks [142]. These scenarios, where high-frequency data streams are involved, typically require more memory and processing power; in return, it may limit Raspberry Pi's effectiveness in handling complex models. Nonetheless, optimizations such as quantization and pruning can be employed to reduce the model size and computational requirements, making it feasible to be on the Raspberry Pi and underscoring its suitability for TinyML deployments [143].

Microcontroller Units (MCUs): are another example of SBC. MCUs are designed to govern operations in embedded systems, where TinyML systems can input direct data from various sensors. It can use a micro, nano level neural network. The system can accommodate an MCU with or without hardware accelerators.Additionally, MCUs allow for real-time data processing and decision-making without relying on cloud resources. They are also supported by an extensive ecosystem of libraries and development tools. One prominent MCU vendor is **STMicroelectronics**, which uses Arm Cortex-M processors as the core CPUs in its STM32 series of MCUs [144]. The STM32 included different models to meet the most demanding balance between the computation performance and the Power requirements for tinyML applications. For instance, the STM32H7 series offers high performance based on the 32-bit Arm Cortex-M7 core, running at up to 600 MHz, embedded flash memory of up to 2 Mbytes, and up to 1.4 Mbytes of SRAM. STM32U5 is another series optimized for ultra power-saving down to 16uA/MHz in the active mode and 110nA in the low power mode. It was built based on an Arm Cortex-M33 of 160MHz speed and equipped with up to 4 Mbytes of flash memory and 3Mbytes of SRAM [144].

Microchip is another vendor offering a wide range of MCUs that span from 8-bit to 32-bit architectures [145]. The 8-bit-based MCUs, such as PIC16 and AVRa6 series, are suitable for basic control and data acquisition applications. The 16-bit-based MCUs, like the 16-bit PIC24F MCU series, are affordable solutions for many applications. These MCUs are equipped with processing up to 70MHz, nonvolatile memory 1024Kbyte, and SRAM up to 52Kbyte. For applications demanding higher performance, microchip assembled the 32-bit MCUs, like the PIC32 family, operating up to 300MHz, with memory of 8192KB and 1024KB RAM. These 32-bit based MCUs provide the performance and functional capabilities to meet the require-

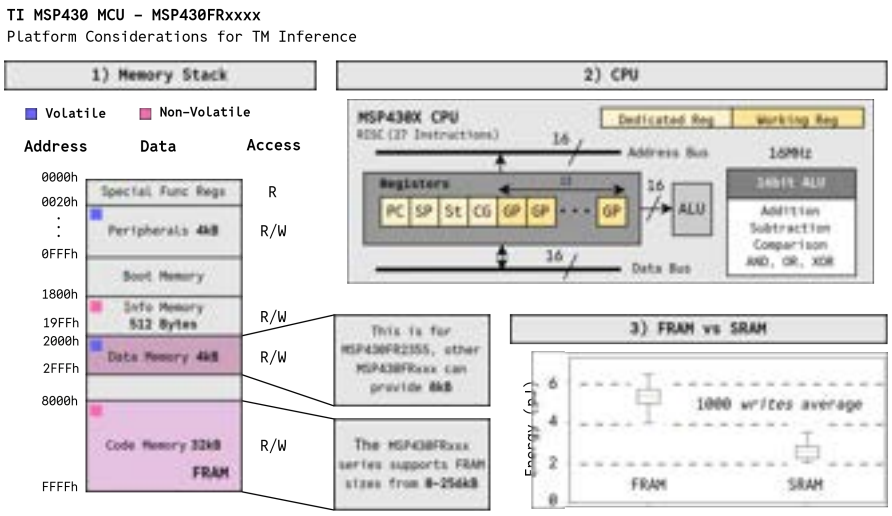


Fig. 21: Stylized MSP board.Stylized overview of the compute and memory capabilities of the MSP430FRxxxx series (FR indicates it uses FRAM).
1) Memory Stack shows the contiguous memory segments that can be traversed through the CPU's stack pointer (SP).
2) CPU shows the MSP430X CPU, a 16bit CPU with limited ALU capability running at around 16MHz. It is connected to 16bit address and data busses. It has 4 dedicated registers: Program Counter (PC), SP, Status Register (St) and Constant Generator (CG). It also has 12 General Purpose registers (GP).
3) FRAM vs SRAM shows the average energy expense of writing to FRAM vs SRAM memory.

ments of modern electronics from general-purpose embedded control to secured IoT solutions. **Texas Instruments (TI)** is another manufacturer supplying the 16-bit MSP430 MCUs to provide affordable solutions for all applications [146]. MSP430 is processing up to 25MHz and has installed 512Kbyte and up to 66Kbyte flash and SRAM memory onboard, respectively as shown in Fig 21. MSP430 is supported by a design resources software called MSP430Ware software and is often used in sensor-based TinyML tasks like anomaly detection [147].

Single Board Computers represent a significant advancement in computing, combining all essential components on a single board for versatile and cost-effective applications. As technology advances, the capabilities of SBCs continue to expand, reinforcing their pivotal role in modern computing. The evolution of the Raspberry Pi reflects the impact of accessibility and innovation on computing. It has transcended its initial function in education to evolve a highly flexible platform for various applications, including TinyML.

Moreover, the various vendors offer specialized MCUs tailored to different applications to be implemented efficiently on-device processing. The persistent advances in MCU technology are paving the way for more sophisticated and capable TinyML devices.

5.8.3 GPU Accelerators (NVIDIA Jetson)

The NVIDIA Jetson platform is a series of embedded computing devices designed to accelerate AI and ML applications at the edge. These devices integrate powerful GPUs with CPUs, enabling efficient processing of complex neural networks through featuring multi power modes, which allow to switch between performance and energy-saving modes, making it adaptable for TinyML use. The Jetson family includes several models, such as Nano, TX2, and Xavier NX, each tailored for different performance and application requirements, as shown in Table 4. The Jetson platform leverages NVIDIA's CUDA architecture, allowing for parallel processing of neural network computations. Jetson supports popular deep learning frameworks such as TensorFlow, PyTorch, and Caffe, facilitating the development and deployment of machine learning models. Its specialized architecture, support for deep learning frameworks, and high performance make it an ideal choice for a wide range of applications, including robotics, healthcare, environmental monitoring, and real-time object detection and tracking to enable to navigate and interact with their environment effectively [148].

Table 4: Jetson Series Specifications [148]

Jetson Series	**AI** (TOPS)	**GPU**		**CPU**		**Memory**		**Power** (W)
		Core	**Max GHz**	**Core**	**Max GHz**	**GB**	**GB/s**	
AGX Orin	200-275	1792-2048	0.93-1.3	8-12	2.0-2.2	32-64	204.8	15-75
Orin NX	70-100	-	0.765-0.918	6-8	2.0	8-16	102.4	10-25
Orin Nano	20-40	512-1024	0.625	6	1.5	4-8	34-68	7-15
AGX Xavier	30-32	512	1.211-1.377	8	2.0-2.2	32-64	136.5	10-40
Xavier NX	21	384	1.1	6	1.9	8-16	59.7	10-20
TX2	1.26-1.33	256	1.12-1.3	Dual Denver2 + quad Cortex-A57	1.95-2.0	4-8	51.2-59.7	7.5-20
Nano	472	128	0.921	Quad Cortex-A57	1.43	4	25.6	5-10

6 Automation, Libraries, and Toolchains

6.1 MATADOR

The MATADOR is another tool that introduces a solution for automating the design and deployment of Tsetlin machine based accelerators on System-on-Chip FPGA (SoC-FPGAs), addressing the constraints faced in edge computing [149]. The MATADOR framework automates the translation of TM models into Register Transfer Level (RTL) designs. Moreover, it significantly facilitates the creation of machine learning inference accelerators for resource constrained environments. One of the key contributions of MATADOR is its ability to outperform state-of-the-art accelerators like Quantized and Binary Neural Networks (QNN/BNN), achieving up to 13.4x faster performance, up to 7x better resource efficiency and 2x lower power consumption. This is made possible by using the inherent sparsity and logic sharing characteristics of TM models, which allows MATADOR to generate compact designs without sacrificing performance. Running on the MNIST dataset, MATADOR achieves a throughput of 3.85 million inferences per second compared to the 954,000 inferences per second of a state-of-the-art BNN, with a corresponding latency reduction to 0.32 µs from 1.047 µs. Additionally, MATADOR reduces resource usage on FPGAs, using only 8,709 Look-up Tables (LUTs) compared to the framework FINN's 11,622 while also lowering dynamic power consumption to 1.29W versus FINN's 1.458W. The MATADOR inference architecture is explained through a block diagram, see Fig. 22 , demonstrating how the accelerator efficiently handles large-scale data inputs by splitting computations into Hard Coded Clause Blocks (HCBs). This framework offers an excellent automated solution for deploying a resource frugal ML models at the edge accelerators.

6.2 LiteRT: The Next Evolution of TensorFlow Lite

LiteRT (short for Lite Runtime) is the new name for TensorFlow Lite (TFLite), marking an expanded vision for Google's high-performance runtime for on-device applications [150]. By supporting multiple frameworks and focusing on the constraints of edge computing, LiteRT enables developers to create sophisticated ML applications that are fast, private, and efficient. LiteRT supports models created not only in TensorFlow but also in PyTorch, JAX, and Keras. This versatility aligns with LiteRT's commitment to enabling developers to deploy models across various platforms while maintaining exceptional performance. LiteRT has been specifically tailored for microcontrollers, enabling machine learning on devices with very limited resources. With a core runtime that fits within 16 KB of memory on an ARM Cortex M3, LiteRT facilitates the deployment of basic models without needing extensive operating system support or dynamic memory allocation. This capability opens the door to smart appliances and IoT devices that can operate intelligently in real-time,

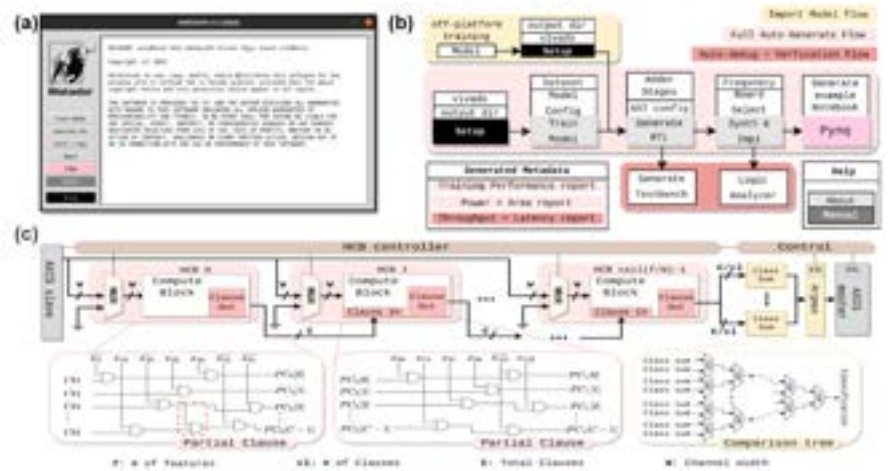


Fig. 22: Matador Design Flow. (a) Matador's GUI. (b) the design generating flow. (c) Block diagram of the generated MATADOR inference accelerator architecture.

all while preserving user privacy by keeping data local. LiteRT is optimized for on-device machine learning by addressing the following constraints:

- **Latency:** Executing ML models directly on the device.
- **Privacy and Security:** User data remains on the device, where LiteRT operates without an internet connection.
- **Binary Size:** The framework is designed to reduce both model and data sizes on resource-constrained devices.
- **Power Efficiency:** LiteRT promotes efficient inference, conserving power and extending device battery life.
- **Multi-Framework Compatibility:** LiteRT offers conversion and optimization tools for transforming models from TensorFlow, PyTorch, and JAX into the efficient FlatBuffers format (.tflite). This adaptability allows developers to leverage state-of-the-art models regardless of their initial training framework.
- **Integrate the Model:** LiteRT provides APIs in multiple programming languages (Java/Kotlin, Swift, Objective-C, C++, Python) for easy integration into applications across various platforms.
- **Pre-Trained Models:** LiteRT offers a rich library of pre-trained models, enabling quick deployment of machine learning functionalities for tasks like Vision APIs (ImageClassifier, ObjectDetector, ImageSegmenter, ImageSearcher, ImageEmbedder), Natural Language (NL) APIs (NLClassifier, BertNLClassifier, BertQuestionAnswerer, TextSearcher, TextEmbedder), Audio APIs (AudioClassifier), and more. Furthermore, LiteRT utilizes delegates to enhance performance through hardware acceleration, leveraging on-device accelerators such as GPUs, CPUs, NPU, and DSPs.

6.3 Edge Impulse

Edge Impulse is a powerful cloud-based platform designed to facilitate the development of machine learning models specifically for TinyML-targeted edge devices. With support for various sensors and hardware platforms, Edge Impulse enables developers to create intelligent applications that can process data directly on edge devices, ensuring low latency and real-time responses. The platform features a user-friendly interface that guides users through the entire machine learning pipeline, from data collection and preprocessing to model training and deployment. Thousands of embedded developers rely on Edge Impulse to run critical ML projects across millions of data samples. The platform addresses the significant challenge of limited memory on Arm Cortex-M MCUs, which typically offer less than 128K of RAM, making it difficult to build, test, and deploy embedded ML models. Edge Impulse has introduced the Edge Optimized Neural (EON) Compiler to overcome this limitation. The EON Compiler accelerates embedded ML code to operate with 25-55% less RAM and up to 35% less flash memory, with the same accuracy as TensorFlow Lite for Microcontrollers. The EON Compiler compiles neural networks to C++. This process simplifies the code, reduces device power consumption, and saves valuable development time [151].

6.4 uTensor:

uTensor is an open-source, lightweight machine learning inference framework optimized for resource-constrained environments, particularly microcontrollers, and IoT-edge devices. Built on top of TensorFlow and Mbed, uTensor is designed to enable rapid prototyping and deployment of machine learning models in devices with minimal computational resources. It converts the pre-trained model from frameworks like Keras into compact, self-contained C++ code suited for deployment on embedded platforms such as Arm-based microcontrollers, where the framework requires as little as 2 KB of memory for its runtime. uTensor's architecture supports convolutional neural networks and pooling layers and utilizes quantization techniques to reduce model size. Its lightweight and flexible design plays a role in advancing the TinyML ecosystem [152].

6.5 The Embedded Learning Library (ELL):

ELL is a framework developed by Microsoft that enables the design and deployment of intelligent machine learning models on resource-constrained platforms such as Raspberry Pi and Arduino. ELL allows models to run on devices without requiring cloud access, making it ideal for applications that demand real time and low latency processing. It serves as a powerful tool for developers, empowering them to create

AI-driven solutions on small, single-board computers. ELL functions as a cross compiler, written in C++ with an optional Python interface to generate machine code on a PC that can be executed on embedded devices. It currently supports tasks such as image and audio classification. Though still in active development, ELL is its focus on model optimization and size reduction ELL toward AI for embedded systems [153].

6.6 STM32Cube.AI:

STM32Cube.AI is an optimization tool to deploy AI models into resource constrained devices. It automatically converts the pre-trained neural networks and classical machine learning models into optimized C code for STM32 microcontrollers. STM32Cube.AI supports deep learning frameworks like Keras and TensorFlow Lite, as well as models from PyTorch and MATLAB using the ONNX format, as shown in Fig. 23. It supports various scikit-learn models such as (isolation forest, support vector machine (SVM), and K-means) as well as quantized neural networks. Moreover, the relocatable option for model updates makes STM32Cube.AI an essential resource-efficient tool for developers. STM32Cube.AI's optimizations ensure that complex neural networks can run efficiently, making it suitable for real-time applications in areas such as IoT, predictive maintenance, and autonomous systems [154].

Fig. 23: STM32Cube.AI framework for deploying AI on STM32 MCUs[154].

6.7 Other Tools:

include NanoEdge AI Studio [155], which facilitates optimal library selection and performance testing before edge deployment, supporting STM32 Nucleo-32 and Ar-

duino Nano 33 IoT boards for anomaly detection and classification tasks. PyTorch Mobile [156] enables executing ML models on edge devices, from the training to deployment within the PyTorch ecosystem, while supporting privacy features like federated learning. Currently in beta, it provides APIs for preprocessing, tracing, and scripting, and integrates XNNPACK and QNNPACK for optimized performance on iOS, Android, and Linux. μTVM extends TVM's compiler for ARM microcontroller targets, enabling on-device autotuning and supporting standalone model inference on various hardware, including STM Nucleo-F746ZG and nRF 5340 Development Kit [157].μTVM optimizes deep learning workloads across diverse hardware, while MinUn addresses memory management and efficiency in TinyML applications [158]. Additional tools to explore include Keras [159], OpenVINO [160], ONNX Runtime [161], SODA [162], and Cainvas [163].

7 Conclusion

TinyML is an innovative approach for enabling the execution of machine learning algorithms on devices with restricted resources, such as microcontrollers, which are frequently used in IoT applications. TinyML has a lot of potential, but in order for it to be widely used and be effective, a lot of challenges must be overcome. In this section, we look at some of the challenges, identify future directions, and provide a roadmap.

7.1 Challenges of TinyML

Resource Constraints: One of the most significant challenges in TinyML is the limited computational power, memory, and energy resources available on edge devices. These constraints necessitate the development of lightweight and highly efficient algorithms that can operate within the confines of microcontroller units. These devices typically run at clock speeds as low as 40-100 MHz and often have less than 1MB of flash memory and SRAM. The trade-off between model complexity and performance is a critical consideration, as more sophisticated models often require more resources than are available on typical IoT devices.

Model Compression and Optimization: To fit machine learning models into the constrained environment of TinyML devices, techniques such as model pruning, quantization, and knowledge distillation are widely used. However, these methods often lead to reduced model accuracy. Achieving the right balance between accuracy and model size is still a major area of research. While advances in hardware (e.g., Tensor Processing Units and Neural Processing Units) have helped, the optimization of models for TinyML still presents significant challenges.

Benchmarking and Standardization: To advance the field of TinyML, establishing a comprehensive benchmarking framework is essential. TinyML models, often

deployed in dynamic environments such as wearable devices or remote sensors, face challenges in performance evaluation due to the need for standardized benchmarks. Current benchmarks, designed for more traditional ML models, need to be adapted to TinyML's constraints. These benchmarks must be designed to support multilevel quantization and accuracy and have enough flexibility to accommodate the heterogeneity of accelerators and platforms. Furthermore, standards are crucial to ensure model interoperability, enabling models to be reused across different platforms. It also promotes scientific comparisons and innovation while reducing development time.

Energy Efficiency: Energy consumption is one of the critical bottlenecks for TinyML, especially in battery-powered edge devices. TinyML system must accommodate energy efficiency in its model's features, where it is designed to operate on minimal energy for extended periods. Despite advancements in energy-efficient algorithms and hardware, achieving a balance between low power consumption and high computational performance is still a major challenge. TinyML models require a certain level of energy to maintain the accuracy level of the algorithms. Hence, It is challenging to specify a universal power-management module for various platforms and applications. Research into energy harvesting for low-power accelerators is crucial for making TinyML applications more possible in the real world.

Security and Privacy: Due to the distributed nature of TinyML, where models operate on decentralized devices that often handle sensitive data, security and privacy are major concerns. TinyML systems must implement secure protocols for communication, data handling, and inference without overburdening the device's limited resources. Privacy preserving techniques, such as on-device processing and encrypted model inference, are increasingly necessary to safeguard data and prevent vulnerabilities. However, the limited resources available for this technology are an obstruction.

7.2 Future Directions

Advanced algorithms for more sophisticated algorithms that can operate efficiently on low resource devices should be in future research. This includes exploring alternative ML architectures specifically designed for TinyML applications, such as lightweight CNNs, RNNs, and SNNs in analog computation that are under research for tinyML applications. These architectures should balance between performance and energy efficiency.

Enhancing the capabilities of TinyML through leveraging methods like federated learning and collaborative approaches. Through Federated learning, multiple devices collaboratively learn from distributed data while maintaining privacy, where only the updates are shared while the raw data never leaves the local device. This helps mitigate concerns about data privacy and security breaches. TinyML can not only improve the model accuracy by implementing collaborative learning but also enhance the model robustness when the data may be heterogeneous and noisy.

Additionally, the integration of TinyML with edge computing can expand its capabilities further. Edge computing enables data processing and decision-making to occur close to the source of data, which is crucial for applications that require real time responses, such as smart home devices, autonomous vehicles, and health monitoring systems. By offloading tasks from the cloud to edge devices, this integration reduces latency, improves bandwidth efficiency, and ensures faster response times, which leads to more complex and sophisticated applications.

ML inference demands significant memory support, where memory facing challenges similar to those in general computing. Memristors and emerging non-volatile memory technologies present promising solutions. Despite the inefficiency of Von Neumann architectures for ML, cost constraints might limit us to using small, inexpensive microcontrollers based on this architecture. This requireing enhancing design through improved training methods, datasets, optimizations techniques . Additionally, exploring new computing paradigms, such as in-memory computing, could benefit ML applications but requires more research and development for incorporation into tinyML systems.

Furthermore, interdisciplinary collaboration is critical to unlocking the full potential of TinyML. Bringing together experts in computer science, electrical engineering, and data science can spark innovative solutions to the challenges TinyML faces and broaden its applications across different fields. To broaden the adoption of TinyML, educational initiatives aimed at increasing awareness and understanding of this technology are essential. Developing accessible resources and training programs can empower a new generation of developers and researchers to contribute to the TinyML ecosystem.

References

1. W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: vision and challenges," *IEEE Internet of Things Journal*, vol. 3, pp. 637–646, 2016.
2. T. Meuser, L. Lovén, M. Bhuyan, S. G. Patil, S. Dustdar, A. Aral, S. Bayhan, C. Becker, E. d. Lara, A. Y. Ding, J. Edinger, J. Gross, N. Mohan, A. D. Pimentel, E. Rivière, H. Schulzrinne, P. Simoens, G. Solmaz, and M. Welzl, "Revisiting edge ai: Opportunities and challenges," *IEEE Internet Computing*, vol. 28, no. 4, pp. 49–59, 2024.
3. T. S. Ajani, A. L. Imoize, and A. A. Atayero, "An overview of machine learning within embedded and mobile devices–optimizations and applications," *Sensors*, vol. 21, p. 4412, 2021.
4. V. J. Reddi, B. Plancher, S. Kennedy, L. Moroney, P. Warden, L. Suzuki, A. Agarwal, C. Banbury, M. Banzi, M. R. Bennett, B. Brown, S. Chitlangia, R. Ghosal, S. Grafman, R. Jaeger, M. Lam, D. Leiker, C. Mann, M. Mazumder, D. Pajak, D. Ramaprasad, J. Smith, M. Stewart, and D. Tingley, "Widening access to applied machine learning with tinyml," *Harvard Data Science Review*, 2022.
5. M. E. Monteiro and A. Nardi, "Evaluation of the energy viability of smart iot sensors using tinyml for computer vision applications: a case study," *International Robotics Amp; Automation Journal*, vol. 9, pp. 78–85, 2023.
6. O.-C. Granmo, "The tsetlin machine – a game theoretic bandit driven approach to optimal pattern recognition with propositional logic," 2021. [Online]. Available: https://arxiv.org/abs/1804.01508

7. J. Morales-García, A. Bueno-Crespo, R. Martínez-España, J. Posadas, P. Manzoni, and J. M. Cecilia, "Evaluation of edge computing platforms through tinyml workloads," 2022.
8. A. K. Nejad, M. Jamshidi, and S. B. Shokouhi, "A super-efficient tinyml processor for the edge metaverse," *Information*, vol. 14, p. 235, 2023.
9. G. Donati, F. Zonzini, and L. D. Marchi, "Tiny deep learning architectures enabling sensor-near acoustic data processing and defect localization," *Computers*, vol. 12, p. 129, 2023.
10. A. Karras, A. Giannaros, C. Karras, L. Theodorakopoulos, C. S. Mammassis, G. A. Krimpas, and S. Sioutas, "Tinyml algorithms for big data management in large-scale iot systems," *Future Internet*, vol. 16, p. 42, 2024.
11. H. Ren, D. Anicic, and T. A. Runkler, "How to manage tiny machine learning at scale: an industrial perspective," 2022.
12. J. Morales-García, A. Bueno-Crespo, R. Martínez-España, J. Posadas, P. Manzoni, and J. M. Cecilia, "Evaluation of low-power devices for smart greenhouse development," *The Journal of Supercomputing*, vol. 79, pp. 10 277–10 299, 2023.
13. M. S. Diab and E. Rodriguez-Villegas, "Embedded machine learning using microcontrollers in wearable and ambulatory systems for health and care applications: a review," *IEEE Access*, vol. 10, pp. 98 450–98 474, 2022.
14. E. Njor, J. Madsen, and X. Fafoutis, "A primer for tinyml predictive maintenance: input and model optimisation," *IFIP Advances in Information and Communication Technology*, pp. 67–78, 2022.
15. V. Tsoukas, A. Gkogkidis, E. Boumpa, S. Papafotikas, and A. Kakarountas, "A gas leakage detection device based on the technology of tinyml †," *Technologies*, vol. 11, p. 45, 2023.
16. W. Raza, A. Osman, F. Ferrini, and F. G. B. D. Natale, "Energy-efficient inference on the edge exploiting tinyml capabilities for uavs," *Drones*, vol. 5, p. 127, 2021.
17. S. Jang, J. Park, E. Lee, and S. Lee, "Lightweight and energy-efficient deep learning accelerator for real-time object detection on edge devices," *Sensors*, vol. 23, p. 1185, 2023.
18. S. E. ARSLAN, "Tiny machine learning model for obstacle detection with multi-zone time of flight sensors," *Recent Advances in Science and Engineering*, pp. 9–14, 2023.
19. A. M. Hayajneh, S. A. Aldalahmeh, F. Alasali, H. Al-Obiedollah, S. A. Zaidi, and D. McLernon, "Tiny machine learning on the edge: a framework for transfer learning empowered unmanned aerial vehicle assisted smart farming," *IET Smart Cities*, vol. 6, pp. 10–26, 2023.
20. A. Hayajneh, S. Aldalahmeh, S. A. R. Zaidi, D. McLernon, H. Obeidollah, and R. Alsakarnah, "Channel state information based device free wireless sensing for iot devices employing tinyml," *2022 4th IEEE Middle East and North Africa COMMunications Conference (MENACOMM)*, 2022.
21. Z. Wang, P. Reviriego, F. Niknia, J. Conde, S. Liu, and F. Lombardi, "Adaptive resolution inference (ari): energy-efficient machine learning for internet of things," *IEEE Internet of Things Journal*, vol. 11, pp. 14 076–14 087, 2024.
22. O. Atanane, A. Mourhir, N. Benamar, and M. Zennaro, "Smart buildings: water leakage detection using tinyml," *Sensors*, vol. 23, p. 9210, 2023.
23. O. O. Martins, A. A. Aribisala, M. M. Abdulhamid, I. O. Joshua, O. J. Oluwatobi, and O. R. Samuel, "Development of a facial recognition model using optimized convolution neural network with tiny-ml and fingerprint authentication for safe lock entry," *Dutse Journal of Pure and Applied Sciences*, vol. 10, pp. 311–321, 2024.
24. N. A. Abbas and M. R. Ahmad, "Keyword spotting system with nano 33 ble sense using embedded machine learning approach," *Jurnal Teknologi*, vol. 85, pp. 175–182, 2023.
25. O. D'Souza, S. C. Mukhopadhyay, and Q. Z. Sheng, "Health, security and fire safety process optimisation using intelligence at the edge," *Sensors*, vol. 22, p. 8143, 2022.
26. A. Gudiño-Ochoa, J. A. García-Rodríguez, R. Ochoa-Ornelas, J. I. Cuevas-Chávez, and D. A. Sánchez-Arias, "Noninvasive diabetes detection through human breath using tinyml-powered e-nose," *Sensors*, vol. 24, p. 1294, 2024.
27. R. Sánchez-Iborra, "Lpwan and embedded machine learning as enablers for the next generation of wearable devices," *Sensors*, vol. 21, p. 5218, 2021.

28. S. Chakkor, M. Baghouri, Z. Cheker, A. E. Oualkadi, A. J. E. Hangouche, and J. Laamech, "Intelligent network for proactive detection of covid-19 disease," *2020 6th IEEE Congress on Information Science and Technology (CiSt)*, 2020.
29. I. N. K. Wardana, S. A. Fahmy, and J. W. Gardner, "Tinyml with meta-learning on microcontrollers for air pollution prediction," *Eurosensors 2023*, 2024.
30. B. Krivokapic, S. Tomovic, I. Radusinovic, and A. Jovanovic, "Implementation and performance evaluation of convolutional neural network models for low-power microcontrollers with constrained resources," *2024 28th International Conference on Information Technology (IT)*, 2024.
31. R. M. M. B. P. R. M. N. Pranav Kumar Chaudhary, Aakash Kishore Chotrani, "Ai in fraud detection: evaluating the efficacy of artificial intelligence in preventing financial misconduct," *Journal of Electrical Systems*, vol. 20, pp. 1332–1338, 2024.
32. X. Wang, Y. Tang, and A. G. Quaranta, "Machine learning-driven lending decisions in bank consumer finance," *International Journal of Information Systems and Supply Chain Management*, vol. 17, pp. 1–19, 2024.
33. R. Agrawal, "Effectiveness of machine learning in stock price prediction," *International Journal for Research in Applied Science and Engineering Technology*, vol. 11, pp. 256–262, 2023.
34. L. Liang, J. Chen, and D. Yao, "Switching to profitable outside options under supplier encroachment," *Production and Operations Management*, vol. 32, pp. 2788–2804, 2023.
35. C. Eroglu, B. D. Williams, and M. A. Waller, "Using the pack-and-a-half rule to eliminate backroom inventories in retail operations," *Journal of Business Logistics*, vol. 39, pp. 164–181, 2018.
36. H. Turtiainen, A. Costin, T. Lahtinen, L. Sintonen, and T. Hamalainen, "Towards large-scale, automated, accurate detection of cctv camera objects using computer vision. applications and implications for privacy, safety, and cybersecurity. (preprint)," 2020.
37. M. Piechocki, M. Kraft, T. Pajchrowski, P. Aszkowski, and D. Pieczyński, "Efficient people counting in thermal images: the benchmark of resource-constrained hardware," *IEEE Access*, vol. 10, pp. 124 835–124 847, 2022.
38. C. Huang, X. Sun, and Y. Zhang, "Tiny-machine-learning-based supply canal surface condition monitoring," *Sensors*, vol. 24, p. 4124, 2024.
39. T. Li, J. Luo, K. Liang, C. Yi, and L. Ma, "Synergy of patent and open-source-driven sustainable climate governance under green ai: a case study of tinyml," *Sustainability*, vol. 15, p. 13779, 2023.
40. S. Ye, J. Luo, H. Liang, and S. Zhao, "The optimal control algorithm of the integrated energy system based on demand response," *Journal of Physics: Conference Series*, vol. 2636, p. 012015, 2023.
41. D. Piatkowski, T. Puślecki, and K. Walkowiak, "Study of the impact of data compression on the energy consumption required for data transmission in a microcontroller-based system," *Sensors*, vol. 24, p. 224, 2023.
42. R. Sánchez-Iborra, L. Bernal-Escobedo, J. Santa, and A. F. Skarmeta, "Tinyml-based fall detection for connected personal mobility vehicles," *Computers, Materials Amp; Continua*, vol. 71, pp. 3869–3885, 2022.
43. C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.
44. S. Maheshwari, T. Rahman, R. Shafik, A. Yakovlev, A. Rafiev, L. Jiao, and O.-C. Granmo, "Redress: Generating compressed models for edge inference using tsetlin machines," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 9, pp. 11 152–11 168, 2023.
45. T. Rahman, A. Wheeldon, R. Shafik, A. Yakovlev, J. Lei, O.-C. Granmo, and S. Das, "Data booleanization for energy efficient on-chip learning using logic driven ai," in *2022 International Symposium on the Tsetlin Machine (ISTM)*, 2022, pp. 29–36.
46. C. Contoli and E. Lattanzi, "Energy efficiency of deep learning compression techniques in wearable human activity recognition," *IFIP Advances in Information and Communication Technology*, pp. 102–113, 2023.

47. R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: a whitepaper," 2018.
48. A. Garofalo, Y. Tortorella, M. Perotti, L. Valente, A. Nadalini, L. Benini, D. Rossi, and F. Conti, "Darkside: a heterogeneous risc-v compute cluster for extreme-edge on-chip dnn inference and training," *IEEE Open Journal of the Solid-State Circuits Society*, vol. 2, pp. 231–243, 2022.
49. P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," 2016.
50. T. Chen, Z. Gan, Y. Liu, L. Zhang, and Z. Wang, "Chasing sparsity in vision transformers: an end-to-end exploration," 2021.
51. H. Zhang, L. Liu, H. Zhou, W. Hou, H. Sun, and N. Zheng, "Akecp: adaptive knowledge extraction from feature maps for fast and efficient channel pruning," 2021.
52. L. Prono, P. Bich, C. Boretti, M. Mangia, F. Pareschi, R. Rovatti, and G. Setti, "A multiply-and-max/min neuron paradigm for aggressively prunable deep neural networks," 2024.
53. S. Disabato and M. Roveri, "Incremental on-device tiny machine learning," *Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Thi*, vol. 46, pp. 7–13, 2020.
54. A. Bakar, T. Rahman, A. Montanari, J. Lei, R. Shafik, and F. Kawsar, "Logic-based intelligence for batteryless sensors," in *Proceedings of the 23rd Annual International Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 22–28. [Online]. Available: https://doi.org/10.1145/3508396.3512870
55. R. K. Kushawaha, S. Kumar, B. Banerjee, and R. Velmurugan, "Distilling spikes: knowledge distillation in spiking neural networks," 2020.
56. M. Abbasi, A. Shahraki, J. Prieto, A. G. Arrieta, and J. M. Corchado, "Unleashing the potential of knowledge distillation for iot traffic classification," *IEEE Transactions on Machine Learning in Communications and Networking*, vol. 2, pp. 221–239, 2024.
57. TensorFlow Team, "Weight clustering: Model optimization guide," 2024, accessed: 2024-10-17. [Online]. Available: https://www.tensorflow.org/model_optimization/guide/clustering
58. S. Dhabal, S. Chakraborty, and P. Venkateswaran, "An efficient quadrature mirror filter design and its applications in audio signal processing," *2011 International Conference on Communication and Industrial Application*, 2011.
59. J. Jiang, F. Zhou, S. Ouyang, and G. Liao, "Efficient design of very large-scale dft modulated filter banks using m th band condition," *IET Signal Processing*, vol. 8, pp. 381–391, 2014.
60. Y. Lu, Z. Song, Q. Gao, D. Zhu, and D. Sun, "Bearing fault diagnosis based on multi-band filtering," *IET Science, Measurement Amp; Technology*, vol. 16, pp. 101–117, 2021.
61. T. Devis and M. Manuel, "Hardware-efficient auto-reconfigurable hearing aids using 3-level octave interpolated filters for auditory compensation applications," *Physical and Engineering Sciences in Medicine*, vol. 44, pp. 785–798, 2021.
62. S. Patel, R. Dhuli, and B. Lall, "Analysis of signals via non-maximally decimated non-uniform filter banks," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, pp. 3882–3895, 2019.
63. U. H. Jaid and A. AbdulHassan, "End-to-end speaker profiling using 1d cnn architectures and filter bank initialization," *International Journal of Online and Biomedical Engineering (iJOE)*, vol. 19, pp. 65–81, 2023.
64. S. M. M. Kim and S. Bleeck, "An open development platform for auditory real-time signal processing," *Speech Communication*, vol. 98, pp. 73–84, 2018.
65. L. Kern, "Design and development of a real-time neural processor using the intel 80170nx etann," in *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, vol. 2, 1992, pp. 684–689 vol.2.
66. Y. Sun, N. B. Agostini, S. Dong, and D. Kaeli, "Summarizing cpu and gpu design trends with product data," 2020. [Online]. Available: https://arxiv.org/abs/1911.11313
67. S. Harris and D. Harris, *Digital Design and Computer Architecture: ARM Edition*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2015.

68. S. Churiwala, *Designing with Xilinx FPGAs: Using Vivado*, 1st ed. Springer Publishing Company, Incorporated, 2016.
69. S. Hosseininoorbin, S. Layeghy, B. Kusý, R. Jurdak, and M. Portmann, "Exploring deep neural networks on edge tpu," 2021.
70. "Project — nicsefc.ee.tsinghua.edu.cn," https://nicsefc.ee.tsinghua.edu.cn/project.html, [Accessed 09-10-2024].
71. V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," 2017. [Online]. Available: https://arxiv.org/abs/1703.09039
72. X. Tang, E. Giacomin, B. Chauviere, A. Alacchi, and P.-E. Gaillardon, "Openfpga: An open-source framework for agile prototyping customizable fpgas," *IEEE Micro*, vol. 40, no. 4, pp. 41–48, 2020.
73. D. Koch, N. Dao, B. Healy, J. Yu, and A. Attwood, "Fabulous: An embedded fpga framework," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 45–56. [Online]. Available: https://doi.org/10.1145/3431920.3439302
74. Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. ACM, Feb. 2017. [Online]. Available: http://dx.doi.org/10.1145/3020078.3021744
75. M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, Dec. 2018. [Online]. Available: https://doi.org/10.1145/3242897
76. Y. Zhang, J. Pan, X. Liu, H. Chen, D. Chen, and Z. Zhang, "Fracbnn: Accurate and fpga-efficient binary neural networks with fractional activations," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 171–182. [Online]. Available: https://doi.org/10.1145/3431920.3439296
77. X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.
78. "hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices," 2021. [Online]. Available: https://arxiv.org/abs/2103.05579
79. M. Andronic and G. A. Constantinides, "Polylut: Learning piecewise polynomials for ultra-low latency fpga lut-based inference," in *2023 International Conference on Field Programmable Technology (ICFPT)*. IEEE, Dec. 2023. [Online]. Available: http://dx.doi.org/10.1109/ICFPT59805.2023.00012
80. Y. Umuroglu, Y. Akhauri, N. J. Fraser, and M. Blott, "Logicnets: Co-designed neural networks and circuits for extreme-throughput applications," 2020. [Online]. Available: https://arxiv.org/abs/2004.03021
81. E. Wang, J. J. Davis, P. Y. K. Cheung, and G. A. Constantinides, "Lutnet: Rethinking inference in fpga soft logic," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 26–34.
82. T. Rahman, G. Mao, S. Maheshwari, R. Shafik, and A. Yakovlev, "Matador: Automated system-on-chip tsetlin machine design generation for edge applications," in *2024 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2024, pp. 1–6.
83. A. Shawahna, S. M. Sait, and A. El-Maleh, "Fpga-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019.
84. A. Yazdanbakhsh, S. Kiran, B. Akin, J. Laudon, and R. Narayanaswami, "An evaluation of edge tpu accelerators for convolutional neural networks," 2021.
85. S. Hosseininoorbin, S. Layeghy, M. Sarhan, R. Jurdak, and M. Portmann, "Exploring edge tpu for network intrusion detection in iot," 2021.

86. H. Kwon, C. Hegde, Y. Kiarashi, V. S. K. Madala, R. Singh, A. Nakum, R. Tweedy, L. M. Tonetto, C. M. Zimring, M. Doiron, A. D. Rodriguez, A. I. Levey, and G. D. Clifford, "A feasibility study on indoor localization and multiperson tracking using sparsely distributed camera network with edge computing," *IEEE Journal of Indoor and Seamless Positioning and Navigation*, vol. 1, pp. 187–198, 2023.
87. L. Klein, P. Žmij, and P. Krömer, "Partial discharge detection by edge computing," *IEEE Access*, vol. 11, pp. 44 192–44 204, 2023.
88. Apple, "Machine learning research at apple: Neural engine and transformers," 2024, accessed: 2024-09-19. [Online]. Available: https://machinelearning.apple.com/research/neural-engine-transformers
89. A. Wiki, "Neural engine," 2024, accessed: 2024-09-19. [Online]. Available: https://apple.fandom.com/wiki/Neural_Engine
90. Z. Fredin, J. Zemanek, C. Blackburn, E. Strand, A. Abdel-Rahman, and P. Rowles, "Discrete integrated circuit electronics (dice)," *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020.
91. D. Xu, M. Xu, Q. Wang, S. Wang, Y. Ma, H. Kang, G. S. Huang, X. Jin, and X. Liu, "Mandheling: mixed-precision on-device dnn training with dsp offloading," 2022.
92. M. Abeysinghe, J. Villarreal, T. T. Hahn, and J. D. Bakos, "Automated loop fusion for image processing," 2024.
93. C. Yang, S. Chen, Y. Wang, and J. Zhang, "The evaluation of dcnn on vector-simd dsp," *IEEE Access*, vol. 7, pp. 22 301–22 309, 2019.
94. M. M. H. Shuvo, S. K. Islam, J. Cheng, and B. I. Morshed, "Efficient acceleration of deep learning inference on resource-constrained edge devices: A review," *Proceedings of the IEEE*, vol. 111, no. 1, pp. 42–91, 2023.
95. Z. I. Chowdhury, S. K. Khatamifard, Z. Zhao, S. Resch, M. Razaviyayn, J. P. Wang, S. S. Sapatnekar, and U. R. Karpuzcu, "Spintronic in-memory pattern matching," *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, vol. 5, pp. 206–214, 2019.
96. R. Srinivasagan, M. Mohammed, and A. Alzahrani, "Tinyml-sensor for shelf life estimation of fresh date fruits," *Sensors*, vol. 23, p. 7081, 2023.
97. O. Ghazal, W. Wang, S. Kvatinsky, F. Merchant, A. Yakovlev, and R. Shafik, "Impact: Inmemory computing architecture based on y-flash technology for coalesced tsetlin machine inference," *arXiv preprint arXiv:2412.05327*, 2024.
98. M. L. Gallo, A. Sebastian, R. Mathis, M. Manica, H. Giefers, T. Tůma, C. Bekas, A. Curioni, and E. Eleftheriou, "Mixed-precision in-memory computing," *Nature Electronics*, vol. 1, pp. 246–253, 2018.
99. A. Haj-Ali, R. Ben-Hur, N. Wald, and S. Kvatinsky, "Efficient algorithms for in-memory fixed point multiplication using magic," *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2018.
100. A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," *Nature nanotechnology*, vol. 15, no. 7, pp. 529–544, 2020.
101. O. Krestinskaya, M. E. Fouda, H. Benmeziane, K. El Maghraoui, A. Sebastian, W. D. Lu, M. Lanza, H. Li, F. Kurdahi, S. A. Fahmy *et al.*, "Neural architecture search for in-memory computing-based deep learning accelerators," *Nature Reviews Electrical Engineering*, pp. 1–17, 2024.
102. S. E. Kim, J. Y. Sung, J. D. Jeon, S. Y. Jang, H. M. Lee, S. M. Moon, J. G. Kang, H. J. Lim, H.-S. Jung, and S. W. Lee, "Toward advanced high-k and electrode thin films for dram capacitors via atomic layer deposition," *Advanced Materials Technologies*, vol. 8, no. 20, p. 2200878, 2023. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/admt.202200878
103. S. Yin, Z. Jiang, J.-S. Seo, and M. Seok, "Xnor-sram: In-memory computing sram macro for binary/ternary deep neural networks," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 6, pp. 1733–1743, 2020.
104. G. W. Burr, R. S. Shenoy, K. Virwani, P. Narayanan, A. Padilla, B. Kurdi, and H. Hwang, "Access devices for 3d crosspoint memory," *Journal of Vacuum Science & Technology B*, vol. 32, no. 4, 2014.

105. S. S. Kim, S. K. Yong, J. Kim, J. M. Choi, T. W. Park, H. Kim, H. J. Kim, and C. S. Hwang, "Fabrication of a hole-type vertical resistive-switching random-access array and intercell interference induced by lateral charge spreading," *Advanced Electronic Materials*, vol. 9, 2022.
106. A. Hazra, A. Tripathi, A. Jan, S. Kundu, and P. K. R. Boppidi, "Multiple nano-filaments based efficient resistive switching in tio2 nanotubes array influenced by thermally induced self-doping and anatase to rutile phase transformation," *Nanotechnology*, vol. 32, p. 115201, 2020.
107. M. Sivan, Y. Li, H. Veluri, Y. Zhao, B. Tang, X. Wang, E. Zamburg, J. F. Leong, J. X. Niu, U. Chand, and A. Thean, "All wse2 1t1r resistive ram cell for future monolithic 3d embedded memory integration," *Nature Communications*, vol. 10, 2019.
108. Z. Sun, G. Pedretti, E. Ambrosi, A. Bricalli, W. Wang, and D. Ielmini, "Solving matrix equations in one step with cross-point resistive arrays," *Proceedings of the National Academy of Sciences*, vol. 116, no. 10, pp. 4123–4128, 2019. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.1815682116
109. Q. Wang, Y. Kim, and P. Li, "Neuromorphic processors with memristive synapses: Synaptic interface and architectural exploration," *J. Emerg. Technol. Comput. Syst.*, vol. 12, no. 4, May 2016. [Online]. Available: https://doi.org/10.1145/2894756
110. S. Lee, S. Lee, J. Lee, J. Choi, D. Kwon, S. Hong, and K. Kwon, "Architecture-accuracy co-optimization of reram-based low-cost neural network processor," *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, pp. 427–432, 2020.
111. W. Chen, Z. Qi, Z. Akhtar, and K. Siddique, "Resistive-ram-based in-memory computing for neural network: a review," *Electronics*, vol. 11, p. 3667, 2022.
112. S. Song, S. Zhang, P. Cheng, Y. Gu, Z. Zhang, Y. Lv, D. Yao, L. Wu, and B. Liu, "Performance improvement of phase-change memory cell using alsb3te and atomic layer deposition tio2 buffer layer," *Nanoscale Research Letters*, vol. 8, 2013.
113. Y. Wang, T. Guo, G. Li, T. Li, S. Lv, S. Song, Y. Cheng, W. Song, K. Ren, and S. Zhang, "Sc-centered octahedron enables high-speed phase change memory with improved data retention and reduced power consumption," *ACS Applied Materials Amp; Interfaces*, vol. 11, pp. 10 848–10 855, 2019.
114. H. Sun, L. Chen, X. Hao, C. Liu, and M. Ni, "An energy-efficient and fast scheme for hybrid storage class memory in an aiot terminal system," *Electronics*, vol. 9, p. 1013, 2020.
115. M. Kim, H. Lee, H. Kim, and H. Lee, "Wl-wd: wear-leveling solution to mitigate write disturbance errors for phase-change memory," *IEEE Access*, vol. 10, pp. 11 420–11 431, 2022.
116. X. Fong, S. H. Choday, and K. Roy, "Design and optimization of spin-transfer torque mrams," *More Than Moore Technologies for Next Generation Computer Design*, pp. 49–72, 2015.
117. Y. Park, G. Kil, and Y. Song, "A novel sensing algorithm for spin-transfer-torque magnetic ram (stt-mram) by utilizing dynamic reference," *IEICE Electronics Express*, vol. 9, pp. 153–159, 2012.
118. M. Mayahinia, M. B. Tahoori, M. Perumkunnil, H. Zahedmanesh, K. Croes, T. Marinelli, J. I. Gomez Perez, T. Evenblij, G. S. Kar, and F. Catthoor, "Time-dependent electromigration modeling for workload-aware design-space exploration in stt-mram," *Ieee Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.
119. S. Yang and Y. Ryu, "A buffer management for stt-mram based hybrid main memory in sensor nodes," *Proceedings of the International Conference on Computer, Networks and Communication Engineering (ICCNCE 2013)*, 2013.
120. A. Shafaei, Y. Wang, and M. Pedram, "Low write-energy stt-mrams using finfet-based access transistors," *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, 2014.
121. X. Fong, R. Venkatesan, A. Raghunathan, and K. Roy, "Non-volatile complementary polarizer spin-transfer torque on-chip caches: A device/circuit/systems perspective," *Ieee Transactions on Magnetics*, 2014.
122. R. Gupta, H. Park, Mojumder, and S. Roy, "Layout-aware optimization of stt mrams," 2012.

123. G. Verma, S. Soni, A. Nisar, and B. K. Kaushik, "Multi-bit mram based high performance neuromorphic accelerator for image classification," *Neuromorphic Computing and Engineering*, vol. 4, p. 014008, 2024.
124. Y. Umeki, K. Yanagida, S. Yoshimoto, S. Izumi, M. Yoshimoto, H. Kawaguchi, K. Tsunoda, and T. Sugii, "Stt-mram operating at 0.38v using negative-resistance sense amplifier," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E97.A, pp. 2411–2417, 2014.
125. O. Ghazal, S. Singh, T. Rahman, S. Yu, Y. Zheng, D. Balsamo, S. Patkar, F. Merchant, F. Xia, A. Yakovlev, and R. Shafik, "Imbue: In-memory boolean-to-current inference architecture for tsetlin machines," in *2023 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2023, pp. 1–6.
126. O. Ghazal, G. Maot, T. Lan, J. Ojukwu, F. Xia, A. Yakovlev, and R. Shafik, "Asynchronous control for tsetlin machine with binary memristor-transistor array," in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2023, pp. 1–5.
127. A. Wheeldon, R. Shafik, T. Rahman, J. Lei, A. Yakovlev, and O.-C. Granmo, "Learning automata based energy-efficient ai hardware design for iot applications," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 378, no. 2182, p. 20190593, 2020. [Online]. Available: https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2019.0593
128. S. Kalapothas, M. Galetakis, G. Flamis, F. Plessas, and P. Kitsos, "A survey on risc-v-based machine learning ecosystem," *Information*, vol. 14, p. 64, 2023.
129. P. Platform, "Pulp: Parallel ultra low power," https://pulp-platform.org/, 2024, accessed: 2024-10-02.
130. J. Lei, T. Rahman, R. Shafik, A. Wheeldon, A. Yakovlev, O.-C. Granmo, F. Kawsar, and A. Mathur, "Low-power audio keyword spotting using tsetlin machines," *Journal of Low Power Electronics and Applications*, vol. 11, no. 2, 2021. [Online]. Available: https://www.mdpi.com/2079-9268/11/2/18
131. M. A. Al Jbaar and S. A. Dawwd, "Simd implementation of deep cnns for myopia detection on a single-board computer system," *Eastern-European Journal of Enterprise Technologies*, vol. 5, pp. 98–108, 2023.
132. H. K. Kondaveeti, S. Raman, and P. Raj, "Prototyping with raspberry pi in healthcare domain," *International Journal of Advanced Computer Science and Applications*, vol. 11, 2020.
133. E. Gamess and S. P. Hernandez, "Performance evaluation of different raspberry pi models for a broad spectrum of interests," *International Journal of Advanced Computer Science and Applications*, vol. 13, 2022.
134. Y. Afrianto, V. Mardiansyah, Ritzkal, F. S. Ramadhan, A. D. Batistuta, B. Wulandari, and W. T. Atmojo, "Enhanced approximation through mimd parallel computing: an efficiency analysis utilizing raspberry pi," *Mathematical Modelling of Engineering Problems*, vol. 10, pp. 1657–1664, 2023.
135. I. N. K. Wardana, J. W. Gardner, and S. A. Fahmy, "Optimising deep learning at the edge for accurate hourly air quality prediction," *Sensors*, vol. 21, p. 1064, 2021.
136. S. R. Manalu, J. V. Moniaga, D. A. Hadipurnawan, and F. Sahidi, "Obd-ii and raspberry pi technology to diagnose car's machine current condition: study literature," *Library Hi Tech News*, vol. 34, pp. 15–21, 2017.
137. D. P. Hausherr and D. Berben, "(all-in-one) power supply system for mobile and network-wired raspberry pi-based internet of things applications," *Hardware*, vol. 1, pp. 54–69, 2023.
138. D. Fernández-Cerero, J. Y. Fernández-Rodríguez, J. A. Álvarez García, L. M. S. Morillo, and A. Fernández-Montes, "Single-board-computer clusters for cloudlet computing in internet of things," *Sensors*, vol. 19, p. 3026, 2019.
139. K. H. Fen, "Development of a portable community video surveillance system," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 9, p. 1814, 2019.
140. I. S. M. Isa, C. J. Yeong, and N. L. Azyze, "Real-time traffic sign detection and recognition using raspberry pi," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 12, p. 331, 2022.

141. M. S. R. Krishna, K. Dinesh, and N. S. Shanbog, "Low cost remote monitoring of solar plant through rs485 communication," *International Journal of Innovative Technology and Exploring Engineering*, vol. 8, pp. 3034–3037, 2019.
142. A. Hassan, H. Nahar, W. M. Shah, A. Abd-Aziz, S. A. Sahiran, N. Bahaman, M. R. Ahmad, I. R. A. Hamid, and M. A. B. Sidik, "Performance evaluation of raspberry pi as an iot edge signal processing device for a real-time flash flood forecasting system," *International Journal of Advanced Computer Science and Applications*, vol. 13, 2022.
143. D. Salami, R. Hasibi, S. Palipana, P. Popovski, T. Michoel, and S. Sigg, "Tesla-rapture: a lightweight gesture recognition system from mmwave radar sparse point clouds," *IEEE Transactions on Mobile Computing*, vol. 22, pp. 4946–4960, 2023.
144. STMicroelectronics, "Stm32 32-bit arm cortex mcus," 2024, accessed: 2024-10-17. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html
145. Microchip Technology Inc., "Microchip microcontrollers (mcus)," 2024, accessed: 2024-10-17. [Online]. Available: https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors
146. Texas Instruments, "Msp430 microcontrollers," 2024, accessed: 2024-10-17. [Online]. Available: https://www.ti.com/design-development/embedded-development/msp430-mcus.html
147. A. Bakar, T. Rahman, R. Shafik, F. Kawsar, and A. Montanari, "Adaptive intelligence for batteryless sensors using software-accelerated tsetlin machines," in *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, 2022, pp. 236–249.
148. NVIDIA, "Nvidia jetson for next-generation robotics," 2024, accessed: 2024-10-16. [Online]. Available: https://www.nvidia.com/en-gb/autonomous-machines/embedded-systems/
149. T. Rahman, G. Mao, S. Maheshwari, R. Shafik, and A. Yakovlev, "Matador: Automated system-on-chip tsetlin machine design generation for edge applications," in *2024 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2024, pp. 1–6.
150. G. AI, "Tensorflow lite for microcontrollers," 2024, accessed: 2024-12-19. [Online]. Available: https://ai.google.dev/edge/litert
151. E. Impulse, "Eon compiler," https://docs.edgeimpulse.com/docs/edge-impulse-studio/deployment/eon-compiler, 2024, accessed: 2024-10-02.
152. uTensor, "Hands on embedded machine learning," https://utensor.github.io/website/, 2024, accessed: 2024-10-02.
153. Microsoft, "Embedded learning library (ell): An open source library for embedded ai and machine learning," https://microsoft.github.io/ELL/, 2024, accessed: 2024-10-02.
154. STMicroelectronics, "Free ai model optimizer for stm32," https://stm32ai.st.com/stm32-cube-ai/, 2024, accessed: 2024-10-02.
155. STMicroelectronics, "Nanoedge ai studio: Automated machine learning (ml) tool for stm32 developers," 2024, accessed: 2024-10-17. [Online]. Available: https://www.st.com/en/development-tools/nanoedgeaistudio.html
156. PyTorch, "Pytorch mobile," 2024, accessed: 2024-10-17. [Online]. Available: https://pytorch.org/mobile/home/
157. Apache Software Foundation, "Tvm: Open deep learning compiler stack," 2024, accessed: 2024-10-17. [Online]. Available: https://tvm.apache.org/
158. S. Jaiswal, R. K. K. Goli, A. Kumar, V. Seshadri, and R. Sharma, "Minun: Accurate ml inference on microcontrollers," ser. LCTES 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 26–39. [Online]. Available: https://doi.org/10.1145/3589610.3596278
159. Keras Team, "Keras: The python deep learning api," 2024, accessed: 2024-10-17. [Online]. Available: https://keras.io/
160. Intel Corporation, "Openvino toolkit," 2024, accessed: 2024-10-17. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/whats-new.html
161. ONNX Community, "Onnx: Open neural network exchange," 2024, accessed: 2024-10-17. [Online]. Available: https://onnx.ai/

162. N. B. Agostini, S. Curzel, A. Limaye, V. Amatya, M. Minutoli, V. G. Castellana, J. Manzano, F. Ferrandi, and A. Tumeo, "Soda: An end-to-end open-source hardware compiler for machine learning accelerators."
163. AI Tech Systems, "Cainvas platform," 2024, accessed: 2024-10-17. [Online]. Available: https://cainvas.ai-tech.systems/accounts/login/