# Machine Learning in the Embedded World

Devashish Pisal
Rheinland-Pfälzische Technische Universität Kaiserslautern, Department of Computer Science

*Abstract: This report looks at how machine learning is growing in use within embedded systems, with a close look at the challenges and the ways to solve them when running ML models on devices that have very few resources. The analysis begins with a discussion of the shortcomings of traditional, cloud-dependent approaches such as increased latency, privacy concerns, and dependency on stable network connectivity. To address these issues, the concept of TinyML is introduced; this paradigm allows for machine learning inference directly on low-power microcontrollers and other edge devices. The report further surveys a variety of hardware platforms for embedded machine learning (ML), including conventional Microcontroller Units (MCUs), Single Board Computers (SBCs), as well as specialized ML accelerators like Tensor Processing Units (TPUs) and Neural Processing Units (NPUs). In addition, essential optimization strategies such as quantization and pruning are discussed, highlighting their importance for fitting large models into the restricted memory and computational budgets of embedded systems. The report is concluded by outlining diverse applications of embedded ML across sectors such as agriculture, autonomous vehicles, healthcare, and smart infrastructure.*

## 1 Introduction

Over the past few years, the machine learning (ML) field has seen a lot of growth, transforming all sorts of areas with different machine learning tools. When it comes to embedded ML, usually, this involves collecting data from small devices like sensors used in smart home gadgets or chips in fitness trackers and sending it to the big cloud servers for processing. But relying on the cloud comes with downsides: it can be slow, raising delays when quick answers are needed; the risk of data breach is there, which leads to privacy and security problems; and it needs a steady connection, which isn't always possible. These problems lead to the adoption of edge computing-based embedded ML, even if embedded devices don't have much power or memory. This idea, called TinyML, runs ML models directly on the embedded systems by finding a good balance between energy consumption and accuracy. These systems use ML by running streamlined models on devices like microcontrollers (MCUs), which can handle real-world inputs, like sensor readings. TinyML lets devices work independently, uses less energy, keeps data safer, and handles urgent tasks without relying on the cloud. For example, in farming, TinyML can detect crop diseases through image analysis on low-power sensors, optimizing yields without constant connectivity. In autonomous vehicles, real-time object detection and path planning are possible because of embedded ML. Similarly, in healthcare, wearable devices can monitor vital

signs and predict health issues locally, providing immediate alerts while protecting sensitive user data.
This report explores the issues and potential solutions for implementing machine learning in resource-constrained devices by using traditional as well as specialized hardware with different ML model compression techniques. Section 2 of this report provides background knowledge about machine learning and embedded systems, which is necessary to understand the further details discussed in this report. Section 3 illustrates real-life applications of embedded ML across various fields. Section 4 introduces TinyML terminology. The report then, in Section 5, surveys both traditional and specialized hardware options available for deploying ML models. Section 6 discusses key optimization techniques such as quantization and pruning, which compress models to fit constrained memory and power budgets. Finally, Section 7 outlines related work and future directions in this evolving field.

## 2 Background Knowledge

### 2.1 Embedded Systems

Embedded systems are designed to perform a specific task in larger systems. They typically consist of hardware, software, microcontrollers, microprocessors, input/output interfaces, and sometimes networking capabilities. They can interact with the physical world, respond to the inputs, actuators, or other devices. They can be reprogrammable or have fixed functionality. But they often have limited processing power. These systems are used in many applications, such as electronics, household devices, industrial machines, etc.

#### 2.1.1 Resource-scarce embedded systems

These systems are part of embedded systems, which operate under significant constraints in terms of computational power, memory, and other resources. Because of their constraints, these systems cannot directly implement ML algorithms, which are complex and resource-intensive in nature. Some examples of resource-constrained embedded systems are Internet of Things (IoT) devices, wearable devices, environmental monitoring systems, etc.

#### 2.1.2 Problems with Embedded ML

When deploying machine learning on resource-scarce embedded systems, several significant challenges arise. In the past, data from these devices was processed by sending it to the cloud. However, this method introduces issues with latency, privacy, and security, and it also requires a reliable internet connection. As a result, the demand for on-device processing has grown. However, embedded systems have limited computational power, memory, and energy, which makes running complex machine learning models difficult. Power consumption, execution time, and model performance must all be carefully balanced before these models can be put into action. Furthermore, the TinyML ecosystem faces its own set of hurdles, including a lack of unified frameworks that can be used across different hardware and the absence of standardized open-source datasets suitable for benchmarking and research. These limitations must be overcome before machine learning can be used in more critical applications, like medical devices and control systems [8, 11, 13, 16]

## 2.2 Machine Learning (ML)

It is a part of Artificial Intelligence where models are trained to perform specific tasks without any explicitly programmed rules [13]. The most common tasks are classification, prediction, pattern recognition, and new data generation. Model building takes a few steps. But the most important steps are training/learning and validation. During model building, the dataset is mostly split into two parts. 75% of the data is used for training, and 25% of the data is used for validation [16]. By providing a model with a larger training data set, bias is reduced, and the model learns the underlying pattern. A smaller validation set provides a reliable estimate of model performance without losing too much training data.

### 2.2.1 Training Phase

For the training phase, there are mainly three key learning types :

- *Supervised Learning:* In supervised learning, the data used for training the model is labeled data. This type of learning is mostly used for classification and regression tasks. The algorithms used for supervised learning are linear regression, decision trees, random forest, support vector machines (SVM), k-nearest neighbours, etc.
- *Unsupervised Learning:* In unsupervised learning, the data used for training the model is unlabeled data in order to find hidden patterns in the dataset. This type of learning is mostly used for clustering tasks. The algorithms used for unsupervised learning are k-means clustering, principal component analysis (PCA), Apriori, etc.
- *Reinforcement Learning:* In reinforcement learning, the model learns by trial and error. For each action taken by the model, it is either treated with a reward or a penalty, and the goal of the model is to find a policy with maximum reward, where a policy is nothing but a mapping of actions taken by the model and the reward/penalty received for that action from the environment. The algorithms used for reinforcement learning are Q-Learning, Monte Carlo Tree Search, etc.

In addition to the above-mentioned learning types, the choice of activation functions for neural networks plays an important role. Among multiple activation functions, Rectified Linear Unit (ReLU) is one of the popular choices, which outputs the input directly if it's positive and sets all values to zero. This helps to avoid issues like vanishing gradients while keeping necessary complexity.
In the training phase, the most crucial factor is the dataset used for the model training. It must fulfill some criteria, such as the data must be error-free, well diversified, and relevant to the task. The quantity of the data also plays an important role in model training [8].

### 2.2.2 Validation Phase

In the validation phase, the developer of the model understands how well the model is performing. During this phase, developers may detect problems like underfitting and overfitting with the model. The overall performance of the classification model can be measured by finding the accuracy of the model by using a confusion matrix [16]. Accuracy is the ratio of correctly classified output instances to all output instances classified by the model. The confusion matrix classifies each output into 4 following sets:

True Positive (TP): Correctly classified positive instance.
True Negative (TN): Correctly classified negative instance.
False Positive (FP): Incorrectly classified positive instance.
False Negative (FN): Incorrectly classified negative instance.

After classifying outputs into these sets, accuracy can be calculated with the following formula:

Accuracy $= \frac{TP+TN}{TP+TN+FP+FN}$

For regression models, absolute/square error techniques are used to evaluate the performance of the model [16].

### 2.2.3 Machine Learning Inference

In the world of machine learning, the process of computing output from a given input and trained model is called inference. Historically, intelligent systems relied on a cloud-centric approach where end-devices with sensors would collect data and send it to an edge-device, which then forwarded it to the cloud for processing and machine learning inference. After that, the results were returned to the end-device. However, this method has significant drawbacks as mentioned in Section 2.1.2, including high latency, security and privacy risks, and a complete dependence on a stable internet connection. A new strategy was developed to address these issues: edge computing, in which some or all of the ML model is executed on an edge device that is closer to the data source. Edge-devices, such as Raspberry Pis, are not resource-scarce and can handle complex computations, thus reducing communication overhead and latency. An even more distributed form of this is fog computing, where model layers are split between the end-device, edge-device, and cloud. For real-time and battery-powered applications, running the inference directly on the end device is often the best choice. This approach eliminates the need for wireless communication, a major drain on a device's power, and ensures critical deadlines are met without the unpredictable delays that come with a network connection [16].

## 3 Use Cases of Embedded ML

Embedded machine learning is an evolving field. There are numerous applications of embedded ML in fields like agriculture, autonomous driving, security, safety, etc. In agriculture, embedded ML can be very helpful in both herbal and livestock farming.

In herbal farming, machine learning models are getting used for various use cases like crop disease protection, pest detection, soil health monitoring, automated irrigation systems, weed detection, environmental monitoring, and many more. Not only in herbal farming but also in livestock farming, facial recognition models are customized for recognizing individual animals during feeding and drinking, parasite detection, health monitoring, and disease detection. For all the above mentioned use cases, inference accuracy and energy consumption are far more important than performance speed.
Autonomous driving has always been an interesting topic for research as well as in science fiction. Machine learning has always been one of the biggest driving factors in the research of autonomous driving. In autonomous driving or assisted driving, machine learning can be used for decision making, path planning, driver monitoring, depth estimation, traffic signal recognition, object/obstacle detection, and many more. But as the nature of this application is safety critical, the inference of the model must be as fast and as accurate as possible [5].
In the area of security, embedded ML can be used for intrusion detection by analyzing motion detectors or door sensors. In surveillance cameras, facial recognition can be used to enhance security and to identify potential threats. It can also be used for voice, fingerprint, or iris recognition to provide a secure authentication mechanism for sensitive information. Additionally, it can be used in modern intelligent weapon systems.
Embedded ML can also be used for safety purposes, such as detecting collisions, fire, or falls. It can be very useful for personnel safety in hazardous worksites (e.g, coal mines) by continuously monitoring the environment or by monitoring the percentage of harmful substances in the environment. It can also prevent vehicle accidents, which are caused by faulty components, by using embedded ML for predictive maintenance.
Beyond agriculture, autonomous driving, safety, and security embedded ML has various applications in fields like healthcare, smart city management, drones and robotics etc. In the field of healthcare, real-time health monitoring is only possible because of machine learning, and it is also used for assisting individuals with disabilities. The power of machine learning can also be used for efficient traffic management, real-time tracking of metro passenger volume, and smart waste management. Machine learning is also used in drones and robotics, by helping robots/drones with navigation and object tracking, and some additional applications are marine life recognition, speech and hand gesture recognition, and real-time video analysis for edge computing [5].

## 4 Introduction to TinyML

It is a field at the intersection of hardware (typically consisting of microcontrollers), software, and machine learning. The goal of this paradigm is to enable ML inference on low-power microcontrollers and other resource-constrained edge devices. This paradigm enables deployment of sophisticated ML algorithms on low-power, resource-constrained microcontrollers. The primary concept of TinyML is to minimize the size of large, power-hungry ML models to the extent that they can fit and run on hardware with just a few kilobytes of memory and processing power, measured in milliwatts. There are several ways to reduce the size of ML models; some of them are discussed in this report in Section 6. The core principles of developing and deploying

TinyML applications are low power consumption, small memory footprint, on-device inference, and low latency. In order to compress the model and deploy it on different hardware platforms, many frameworks and libraries have been developed, some of which are discussed in Section **??**. By adoption of TinyML, the following problems introduced in Section 2.1.2 are solved to much extent:

- *Privacy and security problem:* By keeping all data on the device, the risk of data breaches during transmission to the cloud is greatly reduced.
- *Bandwidth cost problem:* The need for constant data streaming to the cloud is eliminated, saving bandwidth expenses.
- *Internet connectivity problem:* Devices can continue to function even without an internet connection.
- *Power consumption problem:* This extends the battery life of devices, making them more practical for long-term deployment.

Despite its immense potential, TinyML also faces challenges like limited computational resources, a lack of standardized and unified frameworks, and the absence of TinyML-focused open-source datasets, which can be used for benchmarking and further academic research [11].

However, the TinyML framework has undergone significant improvements over the past few years. The first open-source TinyML framework for microcontrollers was Arm uTensor. The result of the lack of a unified framework is the use of specialized frameworks. Over the past few years, several frameworks and libraries have emerged for the TinyML ecosystem such as TensorFlow Lite from Google, PyTorch Mobile from Meta, Edge Impulse, Apache TVM, Embedded Learning Library (ELL), and amlearn [11]. To maintain the focus of this report and adhere to length constraints, a discussion of these frameworks is outside its scope.

## 5 Hardware Used for Machine Learning

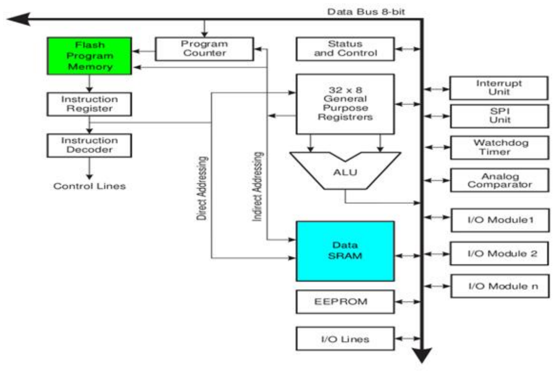

Figure 1: Arduino architecture [1]

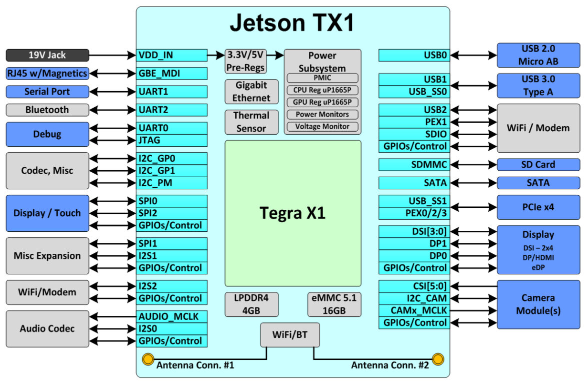

Figure 2: NVIDIA Jetson TX1 block diagram [3]

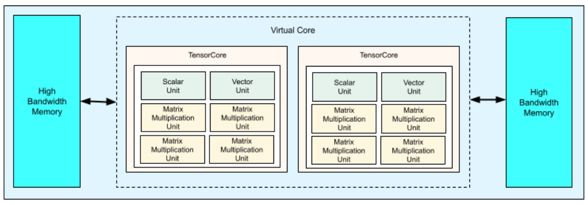
Virtual Core
High Bandwidth Memory
TensorCore
Scalar Unit
Vector Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
TensorCore
Scalar Unit
Vector Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
High Bandwidth Memory

Figure 3: Google's TPU v4 architecture [2]

## 5.1 Traditional Hardware

### 5.1.1 Microcontroller Unit (MCU)

Microcontroller Units are part of embedded systems, designed to execute a single task or a set of functions related to the main task. In the deployment of TinyML algorithms (optimized models only), they are the primary target. The processors used in these MCUs are typically single-core processors, and they come with limited memory and built-in peripherals like Analog-to-Digital converter, Pulse width modulation, etc. MCUs are designed to be power-efficient, which makes them ideal for battery-powered devices and remote deployment. Examples of some of the most widely used MCUs are Arduino UNO, ESP32, and ARM Cortex-M Series. Multiple studies have proven the feasibility of deploying novel machine learning algorithms like ProtoNN, Bonsai, and optimized neural networks on resource-scarce microcontroller units like Arduino UNO and ARM Cortex-M series. The architecture of Arduino UNO is presented in Figure 1. It is based on Harvard architecture, where the program code and program data have separate memory. These implementations highlight the growing potential of TinyML in resource-scarce environments [16].

### 5.1.2 Single Board Computer (SBC)

Single board computers are complete computers on a single circuit board with full functionality and operating systems. Compared to MCUs, they consume more power and are more resource-intensive. Thanks to their multicore CPU architecture, they are designed to run multiple processes in parallel and are capable of handling multiple complex computations and multimedia tasks. The most popular and widely used SBC systems are Raspberry Pis, Banana Pis, ASUS Thinker boards, and ODROID.

### 5.1.3 Graphic Processing Unit (GPU)

- *Problems with Traditional Central Processing Units (CPUs):* Nowadays, most traditional CPUs are based on the Von Neumann architecture, which is not suitable for massive calculations done in machine learning [14]. The key concepts of Von Neumann architecture are storing both data and instruction sets in memory itself, all instructions are executed in a sequential manner, and a common bus system for communication between CPU and memory. This architecture limits the parallel processing capabilities of multicore processors. Furthermore, narrower vector processing units, cache hierarchies, and their instruction sets often lead to underutilization during the highly parallel tasks [14]. All of these factors combined, and the data-, power-hungry nature of ML algorithms make CPUs a bad choice for machine learning tasks.

- *GPUs:* A graphics processing unit is a hardware with thousands of cores in it, which makes it exceptionally good at parallel computing, which is exactly needed for deep learning algorithms. Initially, it was designed for graphics rendering in video games. But because of its massive parallel processing capabilities, the challenges posed by CPUs led to the adoption of GPUs in machine learning. Though GPUs can process massive parallel computations, they usually need a CPU to supply the necessary data and instructions. A well-balanced system with a strong GPU and a good CPU is required for optimal performance while carrying out ML operations. Nowadays, modern GPUs offer cutting-edge technologies like High Bandwidth Memory (HBM) and GDDR6X, which support intense data transfer by reducing the latency [14]. Some examples of modern day GPUs are Jetson Nano, Jetson TX1, Jetson TX2, Jetson Xavier NX, and Jetson Xavier AGX, which integrate GPU and CPU capabilities. The block diagram of the NVIDIA Jetson TX1 is presented in Figure 2. With this System on Chip (SoC) device comes onboard components like 4GB LPDDR4, 16GB eMMC flash, 802.11ac WiFi, Bluetooth 4.0, and Gigabit Ethernet. This SoC accepts DC input voltage between 5.5V and 19.6V. Accepted peripheral interfaces can be seen in the Figure 2.

However, GPUs perform parallel computing tasks much better compared to CPUs; there are certain drawbacks to them, which make them not the ultimate solution for machine learning. Because of their continued reliance on Von Neumann architecture and the nature of machine learning, two main problems arise, which are their significant power consumption and the generation of heat during the computation process. The heat generation problem is mostly resolved by using heat sinks, fans, thermal pastes, and vapor chamber cooling. But the high power consumption problem in GPUs is not yet completely resolved. Because of these limitations of GPUs, the industry has begun to develop specialized hardware accelerators designed specifically for ML workloads [14].

#### 5.1.4 Field-Programmable Gate Arrays (FPGA)

Field-Programmable Gate Arrays are integrated circuits that can be customized by developers after manufacturing to meet specific needs. They offer flexibility similar to CPUs and parallel processing capabilities comparable to GPUs. FPGAs enable developers to create custom hardware circuits programmed using hardware description languages such as Verilog and VHDL. FPGAs are well-suited for low-latency, real-time inference in edge computing and IoT devices. Their key strength is reconfigurability, as it enables the creation of highly energy-saving accelerators for ML algorithms. They excel at parallel processing tasks like matrix multiplication. However, this customization comes with a trade-off: They are more complex to program and resource limitations, such as a finite number of logic gates and memory blocks [7, 14].

## 5.2 Specialized Hardware / ML Accelerators

As a consequence of the recent explosive growth in Machine learning research and limitations of traditional hardware, many tech giants like NVIDIA, Google, and Intel have started developing ML Accelerator chips, which are designed specifically for ML Tasks. Their main focus lies on reduced power consumption, performance, and efficiency. The key features of these specialized ML accelerators are massive parallelism, near-memory processing, high memory bandwidth, energy efficiency, reduced instruction set computing (RISC), and reduced precision arithmetic [14].

### 5.2.1 Tensor Processing Unit (TPU)

Tensor Processing Unit is a custom ML accelerator chip built by Google to speed up machine learning tasks. The architecture of the Google's TPU version 4 is presented in Figure 3. Each TPU v4 chip contains two TensorCores. Each TensorCore has four matrix-multiply units (MXUs), a vector unit, and a scalar unit. This chip is designed with a single purpose: to accelerate deep neural network (DNN) computations, which makes it an Application-Specific Integrated Circuit (ASIC). Although they are especially optimized for the inference phase, they can still speed up the model training phase very significantly. Thanks to their design, they can overcome inefficiencies of traditional multicore CPUs and GPUs when dealing with highly parallel and repetitive math computations, leading to major improvements in power efficiency and speed up [14].
The key features of TPU architecture are:

- *Matrix Multiply Unit (MMU):* This component of the chip is used to perform high-throughput 8-bit multiply-and-add operations and matrix multiplication much faster than general-purpose processors.
- *Systolic Data Flow:* Systolic arrays are used to minimize the read and write from memory, which automatically reduces energy consumption and latency.
- *On-Chip Memory:* As the distance between memory and processing units increases, the latency increases. To tackle this problem, TPUs themselves include memory. Keeping data close to the processing unit is crucial for high performance and efficiency.
- *Reduced Precision Arithmetic:* Instead of using more common 32-bit floating point numbers, TPUs use reduced precision arithmetic (more likely 8-bit integers) to enable fast computation and lower power consumption.
- *Host Integration:* After successful integration of TPUs in the existing server architecture, they act as coprocessors by connecting to the host CPU via standard PCIe I/O bus, where the TPU handles the machine learning tasks, and the remaining general-purpose tasks will be handled by the CPU.

In addition to the above features, TPUs follow a deterministic execution model, which plays a vital role in applications like real-time inference [14].
Google has already deployed its TPUs for many services, like Translate, Photos, and Search, to demonstrate real-world effectiveness. However, some limitations need to be considered. They are

not flexible like general-purpose CPUs, which makes them less suitable for tasks outside neural network computations. TPUs also have a strong dependency on TensorFlow, which makes them software ecosystem dependent. And at the end, the complexity of TPU integration in already established server architecture can also be a big barrier [14].

### 5.2.2 Neural Processing Unit (NPU)

Neural Processing Units are specifically designed for the acceleration of neural network (NN) computations, which makes them ASICs. To handle the unique demands of ML and balance the goal of computational accuracy and efficiency, they are designed. Like TPUs, they operate on a dual-processor approach, which allows the CPU to do precise and general-purpose tasks, and the NPU takes on machine learning specific workloads [14].
NPUs operate on the "Parrot transformation" process. It involves identifying the parts of the program that can be approximated with minimal impact on overall accuracy and then executing them on the NPU. Parrot Transformation involved the following key steps:

- *Code Identification:* The identification of suitable code in a program that has minimal impact on output accuracy.
- *Transformation:* After code identification, the code is transformed into a neural structure by mapping logic and data flow.
- *NPU Execution:* Then this newly created model is sent to the NPU for execution, which is optimized for calculations like matrix multiplication and activation functions.
- *Co-Processing:* The NPU handles the approximate computations, while the CPU executes the part of the program that requires high accuracy.
- *Output Integration:* Once the NPU is done with its approximate calculation, the result is sent back to the CPU and integrated into the main program flow.

Like TPU, the main advantage of NPU is that it gives a significant boost in performance and efficiency by optimally balancing the speed, energy consumption, and accuracy [14]. But they also have some drawbacks. As they are specialized in ML tasks, they cannot be as versatile as CPUs and GPUs. Developing and manufacturing NPUs can be expensive.

**Difference between TPUs and NPUs:**
TPUs are designed for large-scale neural network processing and uses systolic array architecture, which is optimized for matrix multiplication operations and other algebraic operations. On the other hand, NPU chips are designed to accelerate general neural network processing. NPUs can have various architectures like vector processors, systolic arrays, or other custom designs. TPUs are used for large-scale model training in data centers, and NPUs can be used in Edge computing and in other wide range of applications.

## 6 Optimization Techniques

To address the TinyML problems mentioned in Section 2.1.2, such as power consumption, memory issues, security concerns, and limited compute power, several solutions and techniques have emerged over the past few decades. The most popular techniques are Quantization, Pruning, Knowledge-Distillation, Model Architecture Search, Low-Rank Factorization, Efficient Convolutional Kernels, and Hardware-Aware Optimization. These techniques are often used in combination to achieve the best results for TinyML applications. To maintain the focus of this report and adhere to length constraints, we will mainly focus on Quantization and Pruning.

## 6.1 Quantization

The method of mapping a large set of continuous, real-valued numbers to smaller, discrete, and finite numbers is known as Quantization. The primary concept behind quantization is to reduce the number of bits required to represent numerical values, which saves memory, power, and computing resources in neural networks, transforming standard 32-bit floating-point representation to low-precision fixed integer values, such as 8 bits or less. This immediately reduces the model's storage size by roughly 75%, which is critical for MCUs with only kilobytes or megabytes of flash memory. Therefore, it is an active and important research area for the efficient implementation of deep neural networks [6].

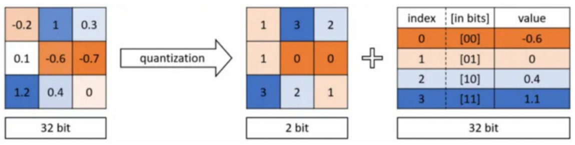

Figure 4: Coversion of 32-bit numbers into 2-bit numbers [4]

Neural networks are heavily over-parameterized, which makes them good candidates for quantization without major negative impact on accuracy. They are also remarkably robust to aggressive quantization and extreme discretization. These unique characteristics have been the main driver for new research into neural network quantization techniques.

1. **Fundamental Concepts**

   There are mainly six types of fundamental quantization techniques, namely uniform, non-uniform, symmetric, asymmetric, static, and dynamic quantization.

   Uniform quantization technique uses evenly spaced quantization levels (intervals between quantized values) and non-uniform quantization uses unequally spaced levels [6]. In uniform quantization, a real-valued input r is mapped to an integer value Q(r) using a scaling factor S and a zero point Z. The scaling factor is determined by a clipping range.

   In symmetric quantization, the used clipping range is centered around zero. This simplifies the quantization by setting the zero point Z to zero, which can reduce computational cost. In asymmetric quantization, the clipping range is not necessarily centered around zero. This approach is often better for neural networks, which use ReLU as an activation function [6].

   In static quantization, the clipping range is pre-calculated by using a small set of calibration inputs. Though it slightly reduces accuracy compared to dynamic quantization, it is still popular because of its negligible overhead. On the other hand, dynamic quantization calculates the clipping range for each activation map at runtime. This method often results in higher accuracy at the cost of very high computational overhead [6].

*Quantization Granularity:* This defines the scope over which a single clipping range is applied. Some main scopes are layerwise quantization, channelwise quantization, groupwise and sub-channelwise quantization.

2. **When can quantization be applied or integrated?**
   Quantization can be integrated during the training process of the machine learning model, or it can also be applied to an already-trained model.

   Quantization-Aware Training (QAT) is applied during training of the model. During the forward pass, a non-differentiable quantization function is used to simulate low-precision arithmetic, and during the backward pass, floating-point weights are updated. Although this process requires high computational power and huge training data, it is effective for regaining any lost accuracy, even with low bit precision [6].

   In Post-Training Quantization (PTQ), a pre-trained model is quantized without any finetuning. Relative to QAT, PTQ is very fast. It is especially applied in scenarios where training data is not made public due to security and privacy reasons. Due to no access to the original training data, the accuracy of PTQ can vary, and it may not always be able to maintain accuracy [6].

3. **Advanced Concepts**

   Some advanced quantization techniques are extreme quantization, mixed-precision quantization, simulated quantization, and integer-only quantization.

   The core concept of extreme quantization is to quantize the model to very low bit precision, such as 1-bit (binary) or 2-bit values. This approach reduces model size, latency during training and learning drastically, which leads to high accuracy degradation unless hyperparameter tuning is performed. This type of quantization is more useful in computer vision. More recently, researchers have attempted to extend this idea to Natural Language Processing (NLP) tasks to address the size and latency problem of state-of-the-art NLP models [6].

   The core idea of mixed-precision quantization is to keep critical layers in higher precision (e.g., 32-bit) and lower-precision quantization (e.g., 4-bit) is applied to less sensitive layers. By applying different bit precision to different layers of a neural network, the balance between accuracy and high efficiency is achieved. Recent work has shown that mixed-precision quantization can provide significant speedups without accuracy degradation [6].

   In a simulated quantization model, parameters are stored in low precision, but while performing actual computations like matrix multiplication, those parameters are dequantized. In integer-only quantization, all parameters are stored as low-bit precision integers, and all operations are performed using low-precision integer arithmetic. Nowadays, many hardware processors, including those from NVIDIA, support fast low-precision arithmetic, which, as a result, boosts inference throughput and latency [6].

## 6.2 Pruning

In pruning, the neural network model is compressed by removing redundant weights or connections. The objective is to speed up inference time without suffering a major performance loss, or to develop a smaller, "sparse" model that may be used on devices with constrained resources, like edge devices. Over the past decade, the number of papers published on pruning increased dramatically, making it a prominent research area in neural network compression [10].

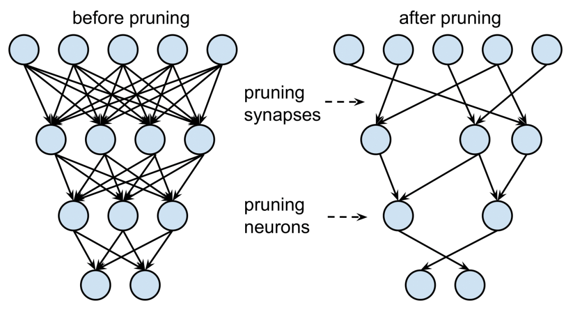

Figure 5: Visualization of sparse network [15]

1. **Types of pruning**
   There are mainly three methods of pruning, namely unstructured pruning, structured pruning, and semi-structured pruning. These methods are categorized based on whether they achieve a universal or specific speedup.

   In unstructured pruning, individual weights are removed from anywhere in the network; this leads to a sparse and irregular model structure. In order to achieve real-world acceleration, unstructured pruning requires specialized hardware and software support [10].

   In structured pruning, entire structured components of the network, like channels, attention heads, or filters, are removed, which results in a smaller model with a regular and narrow structure. Structured pruning does not require any specialized hardware support, and the speedup achieved by this method is universal [10].

   This technique uses both structured and unstructured pruning approaches. Weight removal is done in a specific and predefined pattern to achieve a balance between structural regularity and high accuracy. In the SparseGPT technique, a 2:4 sparsity pattern is used for large language models. 2:4 Sparsity pattern means two out of four consecutive values must be zero. Specialized hardware like NVIDIA's Ampere GPUs supports this pruning technique [10].

2. **When pruning can be applied?**
   Pruning can be performed at different stages in the network's lifecycle. Pruning can be applied before, during, or after model training. Pruning is also possible at run-time.

   In Pruning Before Training (PBT), before any training of the network begins, a subnetwork is identified and pruned from a randomly initialized network. PBT methods like Lottery Ticket Hypothesis (LTH) find the winning subnetworks as early as possible that can be trained from scratch to match the performance of the full, unpruned network [10].

In Pruning During Training (PDT), the network is pruned during the training process. As the training process progresses, weights are pruned based on specific criteria. Methods like Soft filter pruning use the 12 norm of a filter to measure the importance and prune filters with low values. In the Network Slimming method, for each channel, a scaling factor is introduced, and sparsity regularization is used to prune channels with a small scaling factor [10].

Pruning After Training (PAT) is the most adopted pruning technique by the industry. In this technique, the model is trained fully, and after that, the weights are pruned. After pruning, the model is fine-tuned to regain its lost accuracy [10].

Unlike other techniques mentioned above, run-time pruning is a dynamic approach in which, for each individual input, different subnetworks are generated at inference time. The core idea behind this approach is that the complexity needed to process different inputs can vary, so the model needs to activate only certain necessary components of the network, and others can remain inactive [10].

3. **Pruning Criteria**
   Researchers use different pruning criteria or “scoring formulas” to measure the importance of a weight or a filter and prune the least important parts of the network.
   - *Magnitude:* This is a very simple and effective criterion where neural network weights with a smaller magnitude are considered less important and are pruned.
   - *Norm:* In this method, the importance of a group of weights or filters is measured by calculating its norm, and less important filters are pruned. For example, as mentioned above, Soft Filter Pruning uses the 12 norm to identify less important filters.
   - *Loss Change:* This criterion estimates the change in the network’s loss function after removing a specific weight or group of weights. If the estimated change is negligible, then that specific group can be safely pruned. Commonly, it is measured by first-order Taylor expansion [10].

Pruning is often combined with different neural network optimization techniques to achieve better results. It can be combined with quantization to get a more compact and efficient model. It can also be combined with the knowledge distillation process, where the student model is created by a pruning process and retrained by the knowledge distillation process to regain its lost accuracy. Pruning can also be combined with tensor decomposition. Some techniques, like LoSparse, combine low-rank approximations and pruning to compress large models like Transformers [10].

## 6.3 Brief Introduction to Some Other Optimization Techniques

- *Knowledge Distillation (KD):* It is one of the most adopted optimization techniques where a smaller “student” model learns not only the output but also the probability distribution of the output with the help of a larger, high-performing “teacher” model. The goal is to transfer the knowledge from the complex teacher model to the smaller, faster model with the least accuracy loss.

- *Model Architecture Search:* The goal of this approach is to find an efficient neural network design that achieves the best possible performance for a given task. This method can also be used for resource-constrained environments, such as edge devices, to overcome resource constraints.
- *Tensor decomposition:* This method is also known as Low-Rank Factorization. In this method, the number of parameters is reduced by replacing large weight matrices with a set of smaller matrices. This makes the model more efficient for deployment, especially on hardware with limited resources.
- *Efficient Convolutional Kernels:* This technique aims to reduce the computational cost of convolutional operations. Instead of using traditional large kernels, this method employs alternate approaches like separable convolutions and group convolutions to achieve similar results with fewer calculations.
- *Hardware-Aware Optimization:* This method consists of a range of techniques that design or modify neural networks to be more efficient on a specific hardware platform. Hardware-aware optimization involves considering the limitations and capabilities of the target device during the model design and training process because each hardware has unique strengths and weaknesses.

## 7 Related Work and Future Directions

Embedded machine learning has come a long way from initial efforts focused on balancing performance and power consumption to a sophisticated co-design of specialized hardware and software. The development of TPUs and NPUs has turned out to be a pivotal point in embedded ML by proving substantial efficiency gains and speed-ups. In addition to this hardware progress, software frameworks like TensorFlow Lite and model compression techniques, mainly quantization and pruning, have become necessary for deploying ML models on resource-constrained devices. But apart from quantization and pruning, one of the most effective and popular techniques is knowledge distillation. The foundation work on knowledge distillation was established by the authors of [9], who introduced the concept of "soft targets" from a large model to train a compact student model. This core idea is comprehensively surveyed by the authors of [12], who categorized the evolution of distillation into different knowledge types, specialized algorithms, and training schemes.

The future of embedded machine learning is bright and dynamic. On the foundation of hardware-software co-design and model compression, several research directions emerge. The primary focus is on the development of novel algorithms that will be designed for the limitations of resource-scarce MCUs. Intelligent transpilers and synthesizers are also key areas of focus [16]. Another key area of research is to find the optimal combinations of different compression methods, as this field is currently under-researched [6]. Another critical area of development is the co-design of hardware and neural network architecture, especially for FPGAs, to optimize the model and hardware for maximum efficiency [6].
From the perspective of hardware, the future lies in orchestrating heterogeneity, which means

a combination of different hardware accelerators like GPUs, TPUs, and FPGAs to achieve the best performance and energy efficiency. Furthermore, researchers are exploring neuromorphic computing, which aims to mimic the brain's architecture with the help of spiking neural networks (SNN) for ultra-low power consumption and high parallelism [14].
The use of quantum computing is also seen as a key area of research, as it could accelerate both training and inference of models. As these machine learning and TinyML fields evolve, they bring up significant ethical considerations related to their use in surveillance, military, and other controversial applications, which makes transparent and responsible development a necessity [11] [8].

## 8 Conclusion

At the end, it is evident that the conventional approach of depending solely on the cloud for machine learning poses significant limitations, particularly in practical scenarios where every millisecond matters. Challenges such as network delays, security risks, and the requirement for a continuous internet connection are not viable for a genuinely smart ecosystem. This is where the transition to embedded machine learning, or TinyML, becomes essential.
The shift to embedding intelligence within the device is enabled by a blend of factors. There is a growing demand for specialized hardware, as platforms such as TPUs and NPUs are arising to meet the specific needs of ML much more effectively than standard CPUs and GPUs. However, the hardware only constitutes part of the challenge. To enable these models to operate on devices with limited resources, we've needed to identify optimal combinations of optimization methods that provide superior performance while consuming fewer resources.
The integration of specialized hardware and intelligent optimization is not merely a technical milestone; it is driving a new era of applications in multiple domains, ranging from enhancing urban environments to advancing healthcare and transforming autonomous technologies. In the future, the emphasis will probably be on expanding these limits even more. New algorithms tailored for embedded systems, smarter tools for automating model optimization, and the incorporation of advanced technologies like quantum computing are anticipated. Although we must confront the ethical ramifications of this potent technology, the future of decentralized, on-device intelligence appears ready to transform our engagement with technology and our environment.

## References

[1] *Arduino architecture diagram.* Available at https://www.elprocus.com/arduino-basics-and-design/.

[2] *Google's tensor processing unit version 4 architecture image.* Available at https://cloud.google.com/tpu/docs/v4?hl=de.

[3] *NVIDIA's Jetson TX1 chip block diagram.* Available at https://developer.nvidia.com/blog/nvidia-jetson-tx1-supercomputer-on-module-drives-next-wave-of-autonomous-machines/.

[4] *Quantization visualization.* Available at https://xailient.com/blog/4-popular-model-compression-techniques-explained/.

[5] Wei Tang Amin Biglari (2023): *A Review of Embedded Machine Learning Based on Hardware, Application, and Sensing Scheme.* Available at https://www.mdpi.com/1424-8220/23/4/2131.

[6] Zhen Dong Zhewei Yao Michael W. Mahoney Kurt Keutzer Amir Gholami, Sehoon Kim (2021): *A Survey of Quantization Methods for Efficient Neural Network Inference.* Available at https://arxiv.org/abs/2103.13630.

[7] Malaika Mushtaq Amna Shahid (2020): *A Survey Comparing Specialized Hardware And Evolution In TPUs For Neural Networks.* Available at https://ieeexplore.ieee.org/document/9318136.

[8] Eleni Vrochidou Eleftherios Batzolis & George A. Papakostas (2020): *Machine Learning in Embedded Systems: Limitations, Solutions and Future Challenges.* Available at https://ieeexplore.ieee.org/document/10099348/authors.

[9] Jeff Dean Geoffrey Hinton, Oriol Vinyals (2015): *Distilling the Knowledge in a Neural Network.* Available at https://arxiv.org/abs/1503.02531.

[10] Member IEEE Hongrong Cheng, Miao Zhang & Javen Qinfeng Shi (2024): *A Survey on Deep Neural Network Pruning: Taxonomy, Comparison, Analysis, and Recommendations.* Available at https://ieeexplore.ieee.org/document/10643325.

[11] Riku Immonen & Timo Hämäläinen (2022): *Tiny Machine Learning for Resource Constrained Microcontrollers.* Available at https://onlinelibrary.wiley.com/doi/10.1155/2022/7437023.

[12] Stephen John Maybank Dacheng Tao Jianping Gou, Baosheng Yu (2020): *Knowledge Distillation: A Survey.* Available at https://arxiv.org/abs/2006.05525.

[13] Yankit Kumar Kritika Malhotra (2009): *Challenges to implement Machine Learning in Embedded Systems.* Available at https://ieeexplore.ieee.org/document/9362893.

[14] Mrittika Chowdhury Md Sakib Hasan Tamzidul Hoque S M Mojahidul Ahsan, Anurag Dhungel (2024): *Hardware Accelerators for Artificial Intelligence.* Available at https://arxiv.org/abs/2411.13717.

[15] John Tran William J. Dally Song Han, Jeff Pool (2015): *Learning both Weights and Connections for Efficient Neural Networks.* Available at https://arxiv.org/abs/1506.02626.

[16] André G. Ferreira Sérgio Branco & Jorge Cabral (2019): *Machine Learning in Resource-Scarce Embedded Systems, FPGAs, and End-Devices: A Survey.* Available at https://www.mdpi.com/2079-9292/8/11/1289.