# Hardware Architectures For Embedded And Edge AI

Christian Rossi

Academic Year 2024-2025

## Abstract

The goal of this course is to provide a comprehensive understanding of embedded and edge Artificial Intelligence by exploring hardware, software, and algorithmic solutions that support the training and inference of Machine and Deep Learning models on embedded systems and edge computing networks, which are constrained by computation, memory, and energy limitations.

The course will begin with an overview of smart technologies that enable intelligent, autonomous, and distributed embedded and edge applications. It will then dive into a detailed examination of hardware solutions specifically designed for embedded and edge Artificial Intelligence. Students will also gain an in-depth understanding of the frameworks and toolchains, such as Tiny Machine Learning, that support the training and inference of machine and Deep Learning models on these systems. Additionally, the course will cover algorithmic solutions for embedded and edge Artificial Intelligence. A key part of the course will be looking at system-level design for Machine and Deep Learning applications.

# Contents

CHAPTER 1

# Introduction

## 1.1 Introduction

Artificial Intelligence (AI) is a field of computer science focused on developing hardware and software systems capable of performing tasks that typically require human intelligence. These systems can autonomously pursue specific goals by making decisions that were traditionally made by humans.

A key distinction in AI-driven systems lies between smart objects and connected objects. While connected objects primarily send and receive data from the cloud, smart objects analyze data locally, enabling faster decision-making and reducing reliance on constant connectivity.

The definition of AI evolves rapidly, to the point that what was considered AI a decade ago may differ significantly from today's understanding.

AI hardware and software can be categorized similarly to traditional computing environments but are specifically designed to handle AI workloads. In this context, the development environment is often referred to as a framework, platform, or tool, rather than just a conventional programming environment.

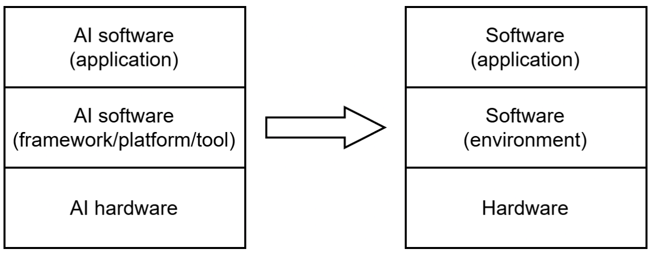

Figure 1.1: Artificial Intelligence stack

The AI stack consists of three main layers:

- *Application*: AI-powered applications running within an IT system.
- *Framework, platform and tools*: programs and libraries that manage physical resources and provide the necessary tools for building AI applications.
- *Hardware*: the infrastructure supporting AI computation, including data centers, edge computing devices, IoT systems, and specialized processors.

## 1.2 Computing continuum

AI hardware ranges from small, low-power devices running on batteries to large-scale, high-performance systems in datacenters. This range represents the computing continuum.

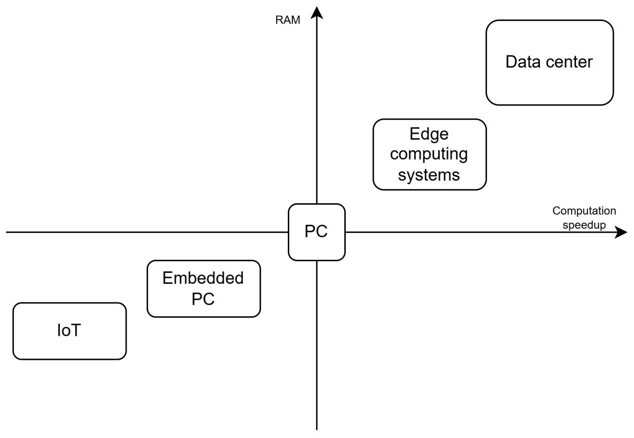

Figure 1.2: Computing continuum

### 1.2.1 Datacenters

Datacenters provide cost-efficient IT infrastructure, high-performance computing capabilities, and instant software updates. Their vast storage capacity ensures data reliability and accessibility, allowing seamless collaboration across devices and locations. Furthermore, by decoupling AI processing from end-user devices, datacenters enable powerful AI applications that are not limited by local hardware constraints.

Datacenters require a constant internet connection. Their reliance on shared infrastructure can introduce privacy and security concerns, while the lack of direct hardware control may limit customization options. Additionally, the high energy consumption of large-scale AI operations raises both environmental and cost concerns. In latency-sensitive applications, delays in data transmission and processing can further impact real-time decision-making.

### 1.2.2 Edge computing systems

Edge computing delivers high computational power with the advantage of distributed processing. By bringing computation closer to where data is generated, it enhances privacy and security while significantly reducing latency in decision-making. However, these systems depend on a stable power supply and often integrate with cloud services to extend their processing capabilities.

By processing data locally, edge computing minimizes the need for constant data transmission, optimizing bandwidth and improving energy efficiency. This approach not only strengthens security and privacy but also enables real-time decision-making and adaptive learning across distributed networks. However, edge devices often operate with limited computing resources, constrained memory, and restricted energy availability. Their design requires careful coordination of hardware, software, and Machine Learning models, adding complexity to development and deployment.

### 1.2.3 Embedded systems

Embedded systems, widely used in AI applications, provide high-performance computing in a compact form. They benefit from the availability of development boards and can be programmed similarly to traditional computers, making them accessible to a broad community of developers. Despite these advantages, they tend to consume relatively high power, and in some cases, require custom hardware design to meet specific application needs.

### 1.2.4 Internet of Things

At the smallest scale, the Internet of Things (IoT) enables AI integration into pervasive, low-cost, battery-powered devices. These systems support wireless connectivity and often include sensing and actuating capabilities, making them essential for smart environments. However, IoT devices face limitations in computing power, energy efficiency, and memory capacity, which can complicate programming and constrain their ability to run advanced AI models.

CHAPTER 2

# Hardware

## 2.1 Introduction

In embedded and edge AI systems, a typical setup includes sensors that capture data from the physical world, software that processes this data, and actuators that execute actions based on computational outcomes. All processing tasks rely on specialized hardware optimized for efficiency and performance.

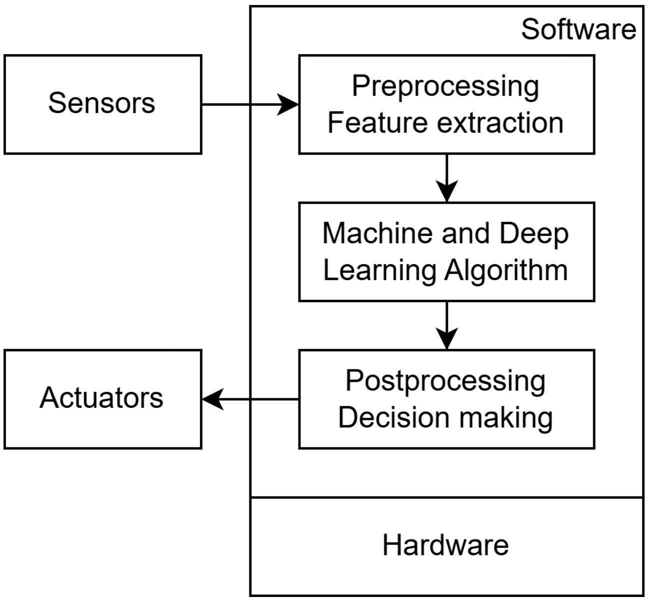

Figure 2.1: AI systems

Embedded systems are computers designed to control and manage the electronics within various physical devices. Embedded software refers to the programs that run on these systems, enabling their functionality.

Unlike general-purpose computers such as laptops or smartphones, embedded systems are typically designed for a specific, dedicated task, ensuring optimized performance, reliability, and energy efficiency for their intended application.

## 2.2 Architecture

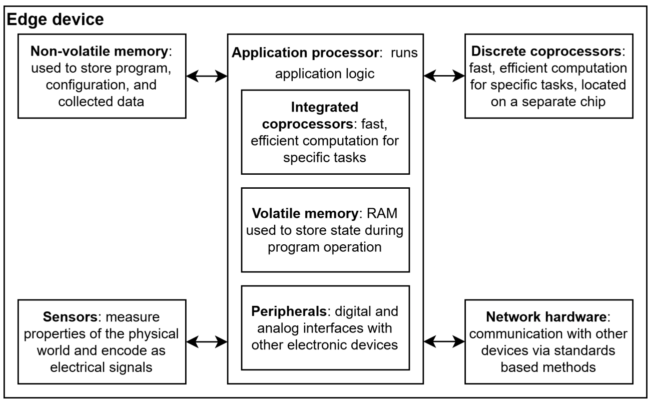
Edge device
**Non-volatile memory**: used to store program, configuration, and collected data
**Application processor**: runs application logic
**Integrated coprocessors**: fast, efficient computation for specific tasks
**Volatile memory**: RAM used to store state during program operation
**Peripherals**: digital and analog interfaces with other electronic devices
**Discrete coprocessors**: fast, efficient computation for specific tasks, located on a separate chip
**Sensors**: measure properties of the physical world and encode as electrical signals
**Network hardware**: communication with other devices via standards based methods

Figure 2.2: Hardware architecture

The hardware architecture of embedded and edge AI systems consists of several key components:

- *Non-volatile memory*: used to store programs, configurations, and collected data. Flash memory is typically used for this purpose, as it retains data even when the system is powered off. It is ideal for storing information that does not change frequently but is slow to read and extremely slow to write.
- *Application processor*: runs the application logic and manages program execution. It includes an integrated coprocessor for efficient computation of specific tasks, volatile memory (RAM) for storing the system state during operation, and various digital and analog peripherals that allow interaction with other electronic components.
- *Discrete coprocessors*: external chips designed for high-speed, efficient mathematical computations. They provide additional processing power for specialized AI workloads that require high performance.
- *Sensors*: measure physical-world properties and convert them into electrical signals for processing. They enable real-time data collection, which is essential for AI-driven decision-making.
- *Network hardware*: ensures communication with other devices using standardized protocols. Reliable connectivity is crucial for data exchange in distributed AI systems.

RAM is often the performance bottleneck in embedded and edge AI systems. It is very fast but consumes significant energy, making efficiency critical in power-sensitive applications. Since it is volatile, data is lost when power is turned off. RAM is also costly and takes up a large physical footprint, impacting the overall design of embedded AI devices.

## 2.3 Sensors and signals

Sensors are used to acquire measurements from the environment or from human interactions. They generate continuous streams of data, which can be used for various AI-driven applications. In addition to sensor data, other sources. Sensors can output data in different formats depending on their purpose and design.

### 2.3.1 Data

**Time series** A time series is a sequence of data points recorded in chronological order. Essentially, it represents observations collected at consistent time intervals. Key factors to consider include the sampling period (time gap between consecutive data points) and bit depth (number of bits used to represent each value).

**Audio** Audio is a specific type of time series, representing sound wave oscillations as they travel through air. Key parameters are: the sampling rate (number of samples taken per second), quantization (bit depth), length (duration of the recording), and the number of channels (mono or stereo). Memory consumption is calculated as:

$$\text{length} \times \text{sampling} \times n \times \text{channel}$$

**Image** Images capture visual information as a grid of pixels, where each pixel represents a specific property of the scene. Key characteristics are: resolution (width and height of the image), bit depth, and channels (RGB or gray-scale). Memory usage is given by:

$$H \times W \times n \times \text{channels}$$

**Video** Videos are sequences of images displayed rapidly to create motion. They share the same structure as images but add the time. The important parameters are the same as the ones of the images with also the frame rate and duration. Memory requirements are determined by:

$$H \times W \times n \times \text{channels} \times \text{frame rate} \times \text{length}$$

### 2.3.2 Sensors

There are thousands of different types of sensors available, each designed to capture specific kinds of data. In the context of embedded and edge AI, sensor technologies can be categorized into six main families:

1. *Acoustic and vibration*: detects sound and mechanical vibrations.
2. *Visual and scene*: captures images, video, and environmental light data.
3. *Motion and position*: measures movement, acceleration, and spatial positioning.
4. *Force and tactile*: detects pressure, touch, and force.
5. *Optical, electromagnetic, and radiation*: measures light, radio waves, and radiation levels.
6. *Environmental and chemical*: monitors temperature, humidity, gases, and other environmental factors.

**Acoustic and vibration** Detecting vibrations is a crucial capability in embedded and edge AI. These sensors allow systems to perceive movement, structural vibrations, and even communication signals from humans and animals at a distance. Acoustic sensors measure vibrations traveling through different media: air (microphones), water (hydrophones), and ground (geophones and seismometers). Since acoustic data is distributed across different frequencies, the sampling frequency plays a key role in ensuring accurate representation for a given application. These sensors typically produce audio data as their output.

**Visual and scene** Visual sensors capture information about the environment without direct contact. These range from tiny, low-power cameras to high-resolution multi-megapixel sensors. Key characteristics of image sensors: color channels, spectral response (infrared sensors), pixel size, resolution, and frame rate. The output of these sensors can be 2D or 3D images or video data, depending on the application.

**Motion and position** Motion and position sensors track movement and spatial positioning in various ways. Examples includes: tilt, accelerometers, gyroscopes, time-of-flight and Global Navigation Satellite System (GNSS). These sensors typically generate time-series data, tracking movement and positioning over time.

**Force and tactile** These sensors help users interact with devices, understand fluid and gas flow, or measure mechanical strain on objects. Example includes buttons, capacitive touch sensors, strain gauges, load cells, flow and pressure sensors.

**Optical, electromagnetic, and radiation** These sensors detect electromagnetic radiation, magnetic fields, and electrical properties. Examples includes: photo-sensors, color sensor, spectroscopy, magnetometers, proximity sensor, electromagnetic field meters and current sensors.

**Environmental, biological, and chemical** These sensors track environmental conditions, biological signals, and chemical presence. Examples includes: temperature, gas, bio signals and chemical sensors. Most of these measurements are typically recorded as time-series data, allowing for trend analysis and predictive insights.

## 2.4 Microprocessor

A microprocessor is a general-purpose processor responsible for running embedded applications. Microcontrollers form the core of many modern, pervasive computing applications. These tiny, cost-effective computers are designed for specific tasks, making them ideal for embedded systems.

Unlike traditional computers, microcontrollers do not require an operating system. Instead, they run firmware (software that is directly executed on the hardware). This firmware includes low-level instructions to manage peripherals and system functions. The defining feature of microcontrollers is that they integrate all essential components into a single silicon chip.

### 2.4.1 Microcontroller

Microcontrollers have a fixed hardware architecture built around a central processing unit. The CPU manages a range of peripherals, providing both digital and analog functionality. Smaller devices typically include both volatile (RAM) and non-volatile (flash) memory on the chip, while more powerful processors may require external memory. Programming is commonly done using low-level languages like Assembly or high-level languages like C.

Microcontrollers are the preferred choice for embedded systems because they integrate essential components like memory and peripherals, reducing the need for additional circuitry. This allows for compact, power-efficient designs that are crucial in space-constrained applications. A microcontroller typically consists of a microprocessor with program (flash) and data

(RAM) memory and various on-chip peripherals. Modern microcontrollers have significantly improved in both performance and efficiency.

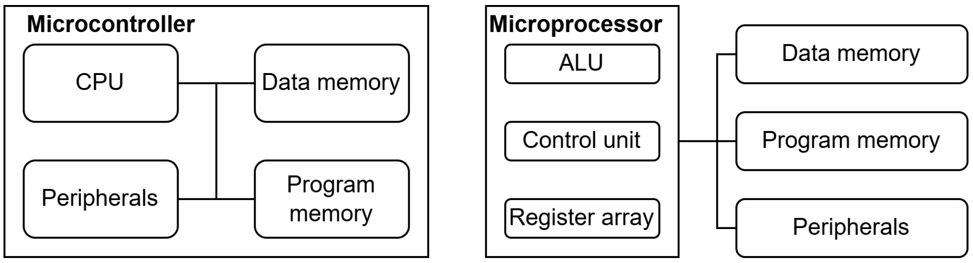

Figure 2.3: Microprocessor and microcontroller

### 2.4.2 System on Chip

A System on Chip (SoC) integrates all the core functionalities of a traditional computing system into a single chip. This includes the CPU, memory, input and output interfaces, and sometimes specialized accelerators for AI, graphics, or signal processing. They runs full operating systems and supports standard development tools and libraries. However, SoC have lower energy efficiency compared to microcontrollers, making them less ideal for ultra-low-power applications, and also more costly.

CHAPTER 3

# Software

## 3.1 Introduction

AI systems in embedded and edge computing are designed to process data efficiently and make real-time decisions.

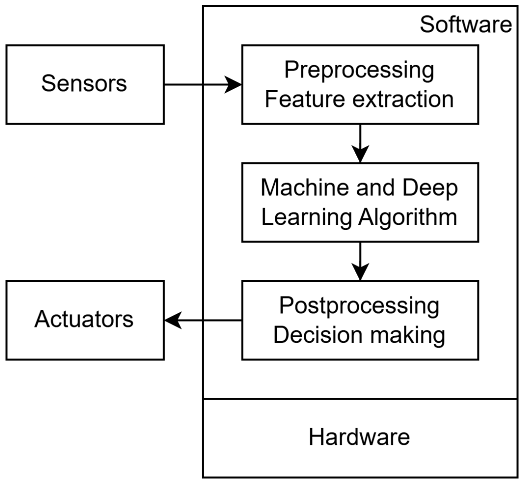
Software
Sensors
Preprocessing Feature extraction
Machine and Deep Learning Algorithm
Actuators
Postprocessing Decision making
Hardware

Figure 3.1: AI systems

The first stage is preprocessing and feature extraction, where incoming data streams from sensors are processed. This step involves segmenting the data into meaningful windows, removing noise, and extracting relevant features to enhance accuracy.

Once the data is preprocessed, Machine and Deep Learning algorithms are applied to analyze and interpret it. These algorithms recognize patterns, make predictions, and generate insights based on the extracted features.

Finally, in the postprocessing and decision-making stage, the system interprets the AI model's output and translates it into meaningful actions. This could involve making decisions, triggering automated responses, or providing feedback to users.

## 3.2 Data preprocessing and feature extraction

The goal of data preprocessing and feature extraction is to convert raw signals into structured data that can be processed by Machine Learning and Deep Learning algorithms. This process involves three key steps:

1. *Signal chucking*: breaking the continuous signal into smaller, manageable chunks of data.
2. *Data processing*: applying digital signal processing techniques to reduce noise and enhance the most relevant parts of the signal.
3. *Feature extraction*: identifying and extracting meaningful patterns or characteristics from the processed signal chunks to be used in model training.

### 3.2.1 Data segmentation

Sensors generate continuous streams of data, which must be divided into smaller segments (windows) for processing. Each window represents a chunk of data that is analyzed by an algorithm to produce meaningful results.

**Definition** (*Latency*)**.** Latency refers to the time required to process a single chunk of data.

Latency plays a crucial role in determining how efficiently an embedded system can process data. Lower latency allows for a higher number of processed chunks per unit of time. However, there is a trade-off:

- Larger windows increase latency but provide more information, often leading to improved accuracy.
- Smaller windows reduce latency but may capture less useful data, potentially affecting performance.

Windows can be overlapping (ensuring no information is lost from the signal), non-overlapping (which may be more computationally efficient but risks missing important details).

**Frame rate** The frame rate defines how frequently the system can acquire and process data, similar to how frame rate applies to image and video streaming.

High latency can hinder real-time analysis by preventing the system from processing new incoming data while still handling previous chunks, potentially leading to data loss. Optimizing both latency and frame rate is essential for effective data segmentation and real-time performance.

### 3.2.2 Data processing

Data preprocessing using digital processing algorithms typically involves three key steps:

1. *Reconstruction of missing data*:
    - *Global filling methods*: filling in missing data based on patterns and trends observed across the entire dataset.
    - *Local filling methods*: estimating missing values using nearby data points. Examples include forward fill, moving averages, and local interpolation techniques.

- *Deletion of affected time periods*: in some cases, time periods with missing data are removed entirely to avoid inaccuracies in analysis.

2. *Resampling*:
   - *Time series resampling*: handling time series data with varying sampling frequencies. In upsampling we increase the sampling rate by replicating or interpolating between timestamps. In downsampling we reduce the sampling rate through sub-sampling. Be cautious of aliasing when changing the sampling frequency.
   - *Image resampling*: adjusting the spatial resolution (pixels per image). In downsampling we reduce the image resolution by decreasing the number of pixels. In interpolation we increase the image resolution by adding pixels based on existing data.
   - *Shape modification for images*: in cropping we trim parts of the image. In resizing we adjust the image dimensions while maintaining or altering aspect ratios.

3. *Filtering*: we can use different types of filter:
   - *Low-pass filter*: retains low frequencies, removing high-frequency noise. Pay attention to the cutoff frequency and frequency response.
   - *High-pass filter*: retains high frequencies, removing low-frequency components. The cutoff frequency and response are important here as well.
   - *Band-pass filter*: keeps a specific range of frequencies, removing those outside the band.

### 3.2.3 Feature extraction

Feature extraction can be applied across various domains and types of data:

- *Time domain*: features like mean, PCA eigenvalues, amplitude, signal-to-noise ratio (SNR), peak decay, and energy can be extracted.
- *Frequency domain*: features such as maximum amplitude, dominant frequency, and peak variance are commonly extracted.
- *Images*: features like edges, corners, blobs, and ridges (curves) are often detected.

**Sensor fusion** In sensor fusion, the goal is to combine data from multiple sensors rather than relying on a single sensor. Each sensor provides unique perspectives and data, and by combining these diverse inputs, we create a more comprehensive and accurate representation of the environment or system.

The features extracted from each sensor are merged to enhance the robustness, accuracy, and reliability of the analysis. These fused features are then used in Machine Learning or Deep Learning models for further processing, ultimately enabling more precise and informed decision-making.

**Normalization and standardization** For more efficient training, data should be normalized or standardized. Features with different scales can negatively impact the performance of Machine Learning models, potentially leading to underfitting.

- *Normalization*: scales data to a range of $[0, 1]$:

$$x' = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}$$

- *Standardization*: centers data around zero mean and with a unitary standard deviation, but does not limit it to a specific range:

$$x' = \frac{x - \mu}{\sigma}$$

  Here, $\mu$ is the mean and $\sigma$ is the standard deviation.

## 3.3 Algorithms

**Functionality** Machine and Deep Learning algorithms can be categorized based on their functionality:

- *Classification*: assigns input data to predefined categories, such as identifying spam emails or recognizing handwritten digits.
- *Regression*: predicts continuous values based on input data, such as forecasting stock prices or estimating house prices.
- *Object detection*: identifies and locates objects within an image or video, commonly used in autonomous driving and security surveillance.
- *Segmentation*: divides an image into meaningful regions, such as medical image analysis or self-driving car lane detection.
- *Anomaly detection*: identifies rare or unusual patterns in data, useful in fraud detection and predictive maintenance.
- *Prediction*: forecasts future outcomes based on historical data, such as weather prediction or demand forecasting.
- *Feature reduction*: reduces the dimensionality of data while preserving essential information, improving efficiency in Machine Learning models.

**Implementation** Machine and Deep Learning algorithms can be categorized based on their implementation approach:

- *Conditional logic*: rule-based systems that use explicit if-then conditions to make decisions, commonly found in expert systems and traditional automation. The advantages of conditional logic are: determinism, efficiency, and there is no need for training data.
- *Machine Learning*: algorithms that learn patterns from data and make predictions without explicit programming.
- *Deep Learning*: a subset of Machine Learning that utilizes multi-layered Neural Networks (NNs) to model complex patterns.

### 3.3.1 Tiny Machine Learning

Tiny Machine Learning (TinyML) focuses on bringing Machine and Deep Learning capabilities to small, low-power devices, particularly in the IoT. This enables real-time data processing directly on the device, reducing reliance on cloud computing and improving efficiency, privacy, and responsiveness.

The key steps in deploying TinyML applications are:

1. *Set up the hardware*: choose a suitable microcontroller or edge device with sufficient processing power and energy efficiency.
2. *Install the software*: set up the necessary development tools, frameworks, and libraries.
3. *Collect data*: gather relevant sensor data to train the model.
4. *Train the model*: develop and optimize a lightweight Machine Learning model tailored for low-power devices.
5. *Build the application*: integrate the trained model into an application that interacts with the device.
6. *Optimize and compile*: convert the application into a format suitable for deployment on the target hardware.
7. *Deploy to the device*: flash the compiled application onto the microcontroller or embedded system.
8. *Test and monitor*: evaluate the performance of the deployed model in real-world conditions and refine as needed.

| **Level** | **Category** |
|---|---|
| **6** | End device training and inference |
| *5* | Edge training and inference |
| *4* | Cloud-edge co-training and inference |
| **3** | Cloud training, on-device inference |
| *2* | Cloud training, edge inference |
| *1* | Cloud training, cloud and edge inference |
| | Cloud training and inference |

CHAPTER 4

# Tiny Machine Learning

## 4.1 Machine Learning

In traditional programming, we provide a computer with both data and a program, and it produces an output based on the instructions we've given. In contrast, with Machine Learning, we feed the computer data and corresponding outputs. The computer then trains itself by adjusting its parameters to minimize a loss function, ultimately producing a trained model that can make predictions.

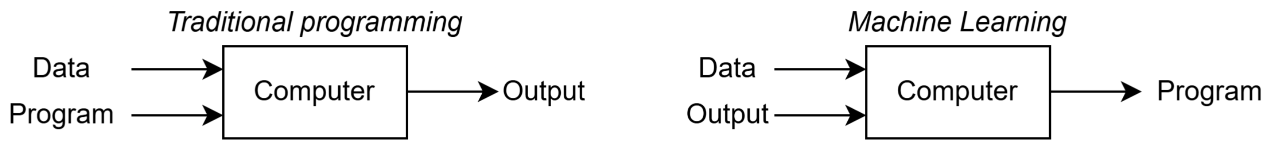

Figure 4.1: Traditional programming and Machine Learning

An essential aspect of Machine Learning is choosing the right model for the task. We aim to select a model that minimizes the error when classifying the training data. Models can belong to various families, such as linear or polynomial models, NNs, and more.

### 4.1.1 Supervised Learning

Let's consider a simple two-dimensional problem. Designing a classifier requires us to identify a function that separates the labeled data points. Some key aspects to focus on include:

- Linear or nonlinear approaches.
- The number of points available or the total number of points.
- The choice of technique for designing the classifier.

**Regression** In nonlinear regression, given a set of noisy data points $(x_i, y_i)$, the goal is to reconstruct the underlying function. This is expressed as:

$$y(x, \mathbf{w}) = \sum_{j=0}^{M} w_j x^j$$

To train the model, we minimize the Residual Sum of Squares with the following formula:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(y(x_n, \mathbf{w}) - t_n)^2$$

Here, $y(x_n, \mathbf{w})$ is the model's output for the input sample $x_n$, $x_n$ represents the input data, and $t_n$ is the target value (the expected output for a given input).

### 4.1.2 Training

The training error isn't always reliable because it only measures the performance of the model on the data used for training. To get a better understanding of how well the model generalizes, we need to split the dataset into a training set and a test set, ensuring that the model isn't tested on the same data it was trained on.

Typically, the test error will be higher than the training error, as the model is being evaluated on unseen data. The amount of data we have plays a crucial role in determining the final quality of the model (more data generally leads to better results).

Additionally, we can introduce a validation set. This allows us to adjust the complexity of the model based on the available data, helping to avoid overfitting or underfitting the model to the training data.

**Model complexity** The complexity of a model, denoted by $M$, plays a crucial role in its performance. If the model is too complex for the available data, it will overfit (it learns the training data too well, including noise, which leads to poor generalization and weak performance on new data). On the other hand, if the model is too simple, it won't capture all the important patterns in the data, leading to high error even on the training set. This is known as underfitting. To achieve the best performance on both training and testing data, we must find an optimal model complexity that balances these two extremes.

**Number of samples** Additionally, the number of samples, $N$ significantly impacts the model's reliability. If we have too few samples, the model may fail to generalize properly, making it difficult to learn the correct patterns.

**Error** The total error in a learning model is composed of three main components:

$$\text{error} = \text{approximation error} + \text{estimation error} + \text{inherent error}$$

Here, we have:

- *Approximation error*: this arises when the chosen model family (or hypothesis space) does not perfectly match the true data-generating process. It represents the error introduced by selecting a model class that may not include the optimal solution.
- *Estimation error*: this occurs when the learning algorithm does not find the best possible model within the chosen hypothesis space. In other words, it's the difference between the best theoretical model within our chosen framework and the actual model we obtain due to limitations in data or training.
- *Inherent error*: this error is unavoidable as it stems from the complexity and noise in the data itself. It depends on the fundamental nature of the learning problem and can only be reduced by improving data quality or redefining the problem.

Reducing both approximation and estimation errors is key to improving model performance, while inherent error is often a fundamental limitation of the problem itself.

### 4.1.3 Validation

To properly evaluate a Machine Learning model, we divide the dataset into three parts:

- *Training set*: used to train the model by adjusting its parameters.
- *Validation set*: used to tune hyperparameters and prevent overfitting.
- *Test set*: used to assess the final model's performance on unseen data.

When evaluating a classifier, we use the confusion matrix, which summarizes the model's performance by comparing predicted and actual class labels:

| | **Predicted positive** | **Predicted negative** |
|---|---|---|
| **Actual positive** | TP (True Positive) | FN (False Negative) |
| **Actual negative** | FP (False Positive) | TN (True Negative) |

Table 4.1: Confusion matrix

Here, the total number of samples is:

$$N = \mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}$$

From this matrix, we can derive key performance metrics:

- *Accuracy*: measures overall correctness but can be misleading for imbalanced datasets:

$$\text{accuracy} = \frac{\mathrm{TP} + \mathrm{TN}}{N}$$

- *Receiver Operating Characteristics*: plots the True Positive Rate against the False Positive Rate. The classifier's performance is represented as a point on this curve. Changing the classification threshold alters this position. A diagonal line corresponds to random guessing, and any performance below this line implies systematically incorrect predictions.

**Model validation** To validate a model we can use various techniques:

- *Apparent Error Rate*: the entire dataset is used for both training and error estimation. However, this method often underestimates the true error due to overfitting.
- *Sample partitioning*: the dataset is randomly split into two disjoint subsets: one used for training, and the other used for evaluation.
- *Leave-One-Out*: improves upon this by iterating over the dataset $N$ times. Each time, one sample is left out for evaluation while the remaining $N - 1$ samples are used for training. The final estimate is obtained by averaging all $N$ iterations.

- *k-fold cross validation*: the dataset is randomly divided into $k$ disjoint subsets of equal size. For each subset, the remaining $k-1$ subsets form the training set, while the reserved subset is used as the evaluation set. The process repeats $k$ times, and the results are averaged. This approach generalizes LOO and reduces variance when $k \ll N$.

By carefully selecting the validation method, we can ensure a more reliable assessment of the model's performance while minimizing bias and variance.

### 4.1.4 Neural Networks

NNs are mathematical models designed to capture relationships in space, time, and the state of individual neurons. Each neuron processes input data through a weighted sum:

$$a_t = \sum_{i=1}^{m} x^i w^i$$

This scalar product measures the affinity between input values and weights. Before activation, it represents the neuron's raw output. An activation function is then applied to introduce non-linearity and determine the final neuron output.

A typical NN consists of an input layer, one or more hidden layers, and an output layer. Training a NN involves minimizing a loss function. For a Feed Forward NN (FFNN), a common loss function is the Residual Sum of Squares:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} (y(\mathbf{x}_n, \mathbf{w}) - t_n)^2$$

Since this function lacks a closed-form solution due to its non-linearity, optimization techniques such as Stochastic Gradient Descent (SGD) are used to iteratively update weights:

$$\mathbf{W}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla \mathcal{L}(\mathbf{w}^{(t)})$$

Here, $\eta$ represents the learning rate, controlling the step size in the direction of the gradient $\nabla \mathcal{L}(\mathbf{w}^{(t)})$. The goal is to find the best possible local minimum, as computing the global minimum is often infeasible.

**Theorem 4.1.1.** *A FFNN with a single hidden layer containing a finite number of neurons can approximate any continuous function defined on compact subsets.*

This theorem implies that, in theory, a properly configured NN can achieve zero approximation error for any target function. However, in practice, finding the exact network that achieves this perfect approximation is extremely challenging.

**Memory demand** The memory required by a FFNN depends on the number of layers and the number of connections within each layer. Every connection between neurons has an associated weight, and each neuron also has a bias, which is another weight that needs to be stored. For a given layer, the total memory demand for weights can be calculated using the following formula:

$$\text{weights} = \sum_{l=1}^{L} N_l \cdot N_{l-1} + N_l$$

Here, $L$ is the total number of layers (excluding the input layer), $N_l$ is the number of neurons in layer $l$, $N_{l-1}$ is the number of neurons in the previous layer. This formula gives the total number of weights that must be stored in memory. If each weight is stored using a specific precision, the total memory demand can be computed by multiplying the number of weights by the storage size per weight.

## 4.2 Deep Learning

Deep Learning is a powerful subset of Machine Learning that focuses on automatically learning the most relevant features from raw data. Unlike traditional Machine Learning methods, which rely on manually extracted features, Deep Learning models take raw input and learn hierarchical representations directly. This capability makes Deep Learning highly effective for complex tasks but also introduces challenges in terms of model complexity and interpretability. While Deep Learning often outperforms traditional Machine Learning, it is not always the best choice (especially when domain expertise allows for efficient manual feature extraction). Additionally, Deep Learning models are computationally more demanding and require significantly more resources to train and deploy.

A typical Convolutional NN (CNN) consists of two main types of layers: convolutional layers and fully connected layers.

### 4.2.1 Convolutional layer

Each convolutional layer typically includes several key components:

- *Convolutional filters* (trainable): these apply spatial operations to the input, with parameters such as filter size, number of filters, and stride.
- *Activation functions*: nonlinear transformations introduce nonlinearity into the model, enabling it to learn complex relationships.
- *Normalization* (trainable) : optional techniques like batch normalization improve training stability and performance.
- *Pooling*: reduces spatial dimensions, helping to control overfitting and computational costs.

Consider an RGB image with dimensions $H \times W \times C$, where $H$ and $W$ represent the height and width of the image, and $C$ denotes the number of channels (3 for RGB, 1 for gray scale). Convolutional filters have dimensions $R \times S \times C$, where $R$ and $S$ are the filter's spatial dimensions, and $C$ matches the number of channels in the input image.

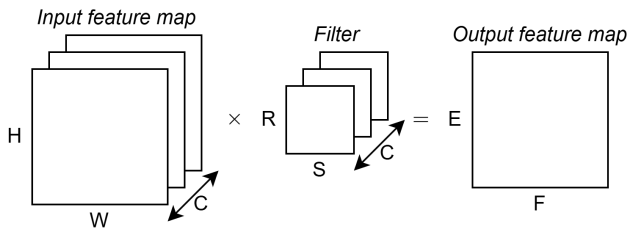

Figure 4.2: Convolutional layer

The output of applying a filter to the image has dimensions $E \times F$, with the number of channels equal to the number of filters used. The exact size of the output depends on the filter dimensions, the stride ($U$), and whether padding is applied. Specifically, with no padding we have :

$$\begin{cases} E = \frac{H-R+U}{U} \\ F = \frac{W-S+U}{U} \end{cases}$$

Each filter has $R \times S \times C + 1$ parameters (including the bias term). Even with a single filter, the number of parameters can grow rapidly, especially for large images.

**Computational demand** The computational cost of a convolutional layer is primarily determined by the number of multiply-and-accumulate (MAC) operations. For a single $3 \times 3$ filter, there are 9 MACs (one for each weight) and a total of $9 + (9 - 1) = 17$ floating-point operations (FLOPs). In general:

$$1 \text{ MAC} \approx 2 \text{ FLOPS}$$

For a convolutional layer with $M$ filters, the total number of MAC operations is:

$$\text{MAC} = E \times F \times R \times S \times C \times M$$

Here, where $E \times F$ is the output size, $R \times S$ is the filter size, $C$ is the number of input channels, and $M$ is the number of filters.

### 4.2.2 Fully connected layer

Fully connected layers consist of densely connected neurons followed by activation functions. These layers are responsible for combining the high-level features extracted by the convolutional layers to make predictions.

In a fully connected layer, each neuron receives input from all neurons in the previous layer. Given an input layer with $H$ neurons and a fully connected layer with $W$ neurons, the total number of parameters is $H \times W + W$, where the additional $W$ accounts for the bias terms.

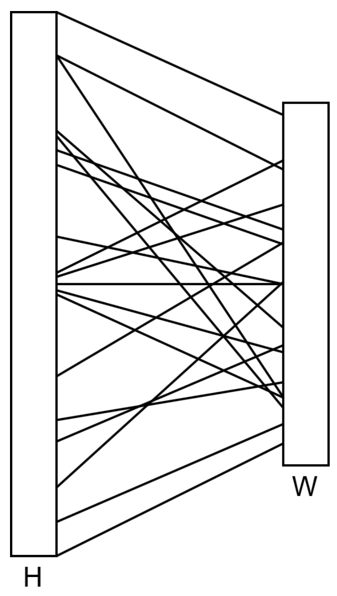

Figure 4.3: Fully connected layer

The computational cost is proportional to the number of MAC operations:

$$\text{MAC} = H \times W$$

### 4.2.3 Memory occupation

Convolutional layers generally occupy less memory than fully connected layers due to their sparse connectivity and parameter sharing. However, they tend to require more MAC operations, making them computationally expensive. Each parameter occupies approximately 4 bytes, so the total memory footprint of a NN is determined by the number of parameters multiplied by this value. Additionally:

- The firmware for the device typically requires around 10 kB.
- Input, output, and intermediate feature maps must also be stored in memory. Their size is given by:
$$H \times W \times C \times 4$$
  This can be a significant bottleneck, especially for large inputs or deep networks.

Modern architectures often reduce the number of fully connected layers while increasing the depth and efficiency of convolutional layers.

### 4.2.4 Tiny Deep Learning

To address the challenges posed by the size and complexity of NNs, we can explore several strategies to make them more efficient and suitable for resource-constrained environments:

- *Redesigning CNN architectures*: develop new, lightweight layers that reduce the overall computational burden while maintaining performance. This involves creating compact network structures that achieve high efficiency without compromising too much on accuracy.
- *Approximate computing*: embrace techniques that trade off a small amount of precision for significant gains in computational speed and memory usage. By allowing minor inaccuracies in calculations, we can drastically improve the efficiency of both training and inference processes.
- *Code optimization for embedded systems*: fine-tune the implementation of Deep Learning models to better suit the hardware constraints of embedded devices. This includes optimizing memory usage, leveraging hardware-specific instructions, and minimizing latency to ensure smooth operation on edge devices.

## 4.3 Tiny Deep Learning architectures

Deep Learning models designed for resource-constrained devices must balance computational efficiency with accuracy. These architectures are optimized to reduce memory usage, computation overhead, and energy consumption while maintaining competitive performance. In this context, three prominent families of architectures have emerged: SqueezeNet, MobileNet, and EfficientNet.

### 4.3.1 SqueezeNet

Introduced in 2016, SqueezeNet was the first CNN architecture explicitly designed for deployment on edge devices. It achieves a remarkable reduction in model size (up to 50 to 500 times fewer parameters than AlexNet) while maintaining comparable accuracy. This makes SqueezeNet highly suitable for distributed training environments, where communication overhead is proportional to model size. Additionally, its smaller footprint reduces the cost of exporting updated models to clients.

The objectives of SqueezeNet are to design a CNN with fewer parameters but equivalent accuracy to AlexNet. To achieve this goal, SqueezeNet employs several innovative strategies:

1. *Replace* $3 \times 3$ *filters with* $1 \times 1$ *filters*: using $1 \times 1$ filters instead of $3 \times 3$ filters reduces the number of parameters by a factor of 9. This also decreases the number of MACs. However, not all $3 \times 3$ filters are replaced, as they are essential for capturing spatial patterns.

2. *Reduce input channels to the* $3 \times 3$ *filters*: to further minimize the number of parameters, SqueezeNet introduces squeeze layers that reduce the number of input channels to $3 \times 3$ filters. This results in fewer $3 \times 3$ filters overall. This is done using squeeze layers.

3. *Downsample late in the network*: delaying downsampling preserves larger activation maps, which can improve accuracy despite using fewer weights. While this increases memory usage, it prevents the loss of critical features in early stages due to overly aggressive dimensionality reduction.

**Fire module** At the heart of SqueezeNet is the fire module, a novel convolutional layer designed to drastically reduce memory and computation demands while preserving accuracy. Each Fire module consists of two stages:

1. *Squeeze layer*: applies only $1 \times 1$ convolutional filters. Reduces the number of input channels to $s$, a hyperparameter chosen during design. The output dimensions is $H \times W \times s$.

2. *Expand layer*: takes the reduced activation map as input. Applies $e$ $3 \times 3$ filters and $e$ $1 \times 1$, producing an output with $2e$ channels. Need $s < 2e$ to maintain computational efficiency. The output dimensions is $H \times W \times 2e$.

#### 4.3.1.1 Architecture

The final example architecture is composed by:

1. *Initial convolutional layer*: processes the input image and prepares it for subsequent Fire modules.

2. *Eight fire modules*: each module progressively increases the number of filters to capture more complex features. Max-pooling is applied after Fire modules 4 and 8 to downsample the feature maps.

3. *Global Average Pooling and softmax*: instead of fully connected layers, SqueezeNet uses GAP followed by a softmax layer for classification. This eliminates the need for dense layers, further reducing the parameter count.

Further techniques, such as sparsity (pruning unnecessary weights) and quantization (reducing precision), can be applied to SqueezeNet to enhance efficiency without significant accuracy loss. The base SqueezeNet model achieves AlexNet-level accuracy while requiring far fewer parameters and MACs.

#### 4.3.1.2 Variants

Several variants of SqueezeNet have been proposed to improve accuracy with minimal additional complexity:

1. *Simple bypass architecture*: adds skip connections around specific fire modules. These connections allow the modules to learn residual functions, improving gradient flow and accuracy. No extra parameters are introduced, making this variant highly efficient.

2. *Complex bypass architecture*: incorporates $1 \times 1$ convolutional layers within the skip connections. The number of filters in these convolutions equals the number of output channels, introducing additional trainable parameters. While slightly more accurate, this variant increases the model's size and computational cost.

Among the variants, the simple bypass architecture provides the best trade-off, offering modest accuracy improvements without adding extra weights.

### 4.3.2 MobileNet

MobileNet is a family of efficient NN architectures designed specifically for mobile and embedded vision applications. It achieves lightweight computation through the use of depth-wise separable convolutions, a streamlined approach that significantly reduces the number of parameters and computational cost while maintaining competitive accuracy. This architecture has become a cornerstone for deploying Deep Learning models on resource-constrained devices.

**Depth-wise scalable convolution** The key innovation in MobileNet is the use of depth-wise separable convolutions, which decompose a standard convolution into two simpler operations:

1. *Depth-wise convolution*: applies a single $R \times S$ filter to each input channel independently. Unlike a standard convolution, this step does not combine information across channels. The output dimensions is $H \times W \times C$, where $C$ is the number of input channels.

2. *Point-wise convolution*: combines the outputs of the depth-wise convolution across channels using $M$ $1 \times 1$ filters. This step aggregates channel-wise information, producing the final output feature map.

| **Convolution** | **MAC operations** | **Weights** |
|---|---|---|
| Standard | $R \times S \times C \times E \times F \times M$ | $R \times S \times C \times M$ |
| Depth-wise | $R \times S \times C \times E \times F + C \times M \times E \times F$ | $R \times S \times C + C \times M$ |

By decoupling spatial and channel-wise computations, depth-wise separable convolutions achieve significant reductions in both MAC operations and the number of weights compared to standard convolutions.

#### 4.3.2.1 Architecture

Each block in MobileNet follows a consistent structure:

1. *Depth-wise convolution*: applies a 3×3 filter to each input channel independently. Reduces computational cost by avoiding cross-channel interactions at this stage.
2. *Batch normalization*: normalizes the outputs of the depth-wise convolution to stabilize training.
3. *ReLU activation*: introduces non-linearity to the model.
4. *Point-wise convolution*: combines information across channels using $1 \times 1$ convolutions.
5. *Batch normalization*: normalizes the outputs of the point-wise convolution.
6. *ReLU activation*: applies another non-linear transformation.

### 4.3.3 EfficientNet

EfficientNet represents a breakthrough in the design of efficient CNNs by introducing a novel method for scaling up models. Unlike traditional approaches that focus on increasing either depth (layers), width (filters), or resolution individually, EfficientNet employs compound scaling, which scales all three dimensions simultaneously while maintaining a balanced relationship between them.

The key idea behind compound scaling is to scale depth ($\alpha^\phi$), width ($\beta^\phi$), and resolution ($\gamma^\phi$) with constant ratios:

$$\alpha\beta^2\gamma^2 \approx 2 \qquad \alpha, \beta, \gamma \geq 1$$

Here, $\alpha$, $\beta$, and $\gamma$ are constants that control how each dimension is scaled, and $\gamma$ is a user-specified coefficient that determines the extent of scaling (how many more resources are available). EfficientNet's compound scaling is implemented in two steps:

1. *Determine scaling coefficients ($\alpha, \beta, \gamma$)*: start with a baseline network and assume twice the available computational resources (double $\phi$). Perform a small grid search to find optimal values of $\alpha$, $\beta$, and $\gamma$, subject to the constraint $\alpha\beta^2\gamma^2 \approx 2$. Fix these coefficients as constants for subsequent scaling.
2. *Scale the baseline network*: use the fixed scaling coefficients to scale the baseline network to different sizes by varying $\phi$. Increasing $\phi$ corresponds to enlarge the model, achieving higher accuracy at the cost of increased computational resources.

## 4.4 Approximate computing

When architectural simplifications alone are insufficient to meet resource constraints, approximate computing techniques can be employed. These methods trade off some computational precision to achieve reductions in memory usage and computational load. Common approaches include pruning, quantization, and distillation.

A NN typically consists of $l$ layers, each defined by a function $\phi_{\boldsymbol{\theta}_i}$, where $\boldsymbol{\theta}_i$ represents the parameters of the layer. Based on this structure, two primary approximate computing techniques can be applied:

- *Precision scaling*: this technique reduces the memory footprint of a Convolutional NN by lowering the precision of the weights. For instance, representing weights with fewer bits decreases memory requirements while maintaining acceptable performance.
- *Task dropping*: this approach minimizes computational load and memory usage by selectively skipping the execution of certain tasks within the processing chain. These tasks may correspond to specific layers or neurons whose omission has minimal impact on the overall output quality.

### 4.4.1 Precision scaling

Precision scaling is achieved through quantization, which reduces the precision of weights and activations in NNs. Quantization can be categorized into three main types:

- *Linear quantization*: this approach uses a uniform distance between quantization levels. Linear quantization is widely used due to its simplicity and effectiveness.
- *Logarithmic quantization*: the distance between quantization levels varies logarithmically.
- *Data-driven quantizer*: levels are determined or learned from the data.

Quantization can target either weights (to reduce storage requirements) or activations (typically during inference). Fixed quantization applies the same quantization scheme across all layers, while variable quantization tailors the quantization mechanism to specific layers, filters, or channels. The first and last layers of a network are often the most critical to quantize carefully, as they significantly impact accuracy.

**Performance** For CNNs, quantizing convolutional layers to 8 bits and fully connected layers to 4 bits typically results in negligible accuracy loss for weights. Activations, however, are usually quantized to 16 bits, as their quantization has a more pronounced impact on performance. Interestingly, quantization can sometimes act as a regularizer, slightly improving accuracy in rare cases.

In practice, implementing data in 8-bit formats is common. Four 8-bit operations are approximately equivalent to one 32-bit operation within a given clock cycle. An 8-bit fixed-point addition consumes 3.3 times less energy than a 32-bit fixed-point addition and 30 times less energy than a 32-bit floating-point addition. An 8-bit fixed-point multiplication consumes 15.5 times less energy than a 32-bit fixed-point multiplication and 18.5 times less energy than a 32-bit floating-point multiplication.

Extreme quantization techniques push the limits of precision reduction:

- *Binary networks*: weights are restricted to -1 and +1 by computing the sign of the output. Binary Connect uses binary weights, while Binary NNs use both binary weights and activations. These methods incur an accuracy loss of approximately 20%-30%.
- *Ternary networks*: weights are restricted to -w, 0, and +w. Ternary weight networks share the same scale factor for all weights, while trained ternary quantization learns a different scale for each weight. Ternary quantization incurs minimal accuracy loss ($\approx 0.5\%$) and is of theoretical interest.

#### 4.4.1.1 Implementation

A schematic overview of matrix-multiply logic in NN accelerator hardware involves weights $w_{n,m}$ and input values $x_n$. The Multiply-Accumulate (MAC) result is computed as:The Multiply-Accumulate (MAC) result is computed as:

$$A_n = b_n + \sum_m C_{n,m}$$

Here, $C_{n,m}$ is the product of weight $w_{n,m}$ and input $x_n$.

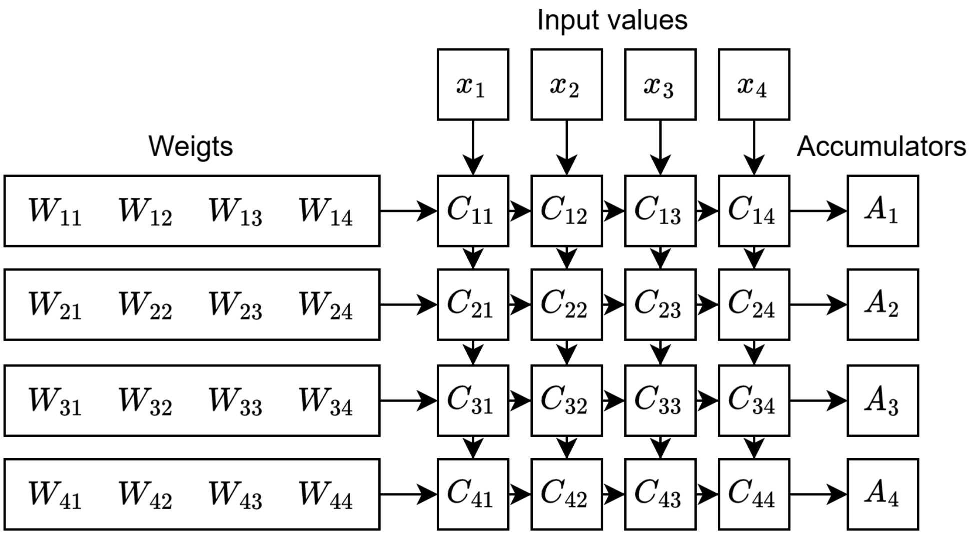
Input values
$x_1$
$x_2$
$x_3$
$x_4$
Weigts
Accumulators
$W_{11}$ $W_{12}$ $W_{13}$ $W_{14}$
$W_{21}$ $W_{22}$ $W_{23}$ $W_{24}$
$W_{31}$ $W_{32}$ $W_{33}$ $W_{34}$
$W_{41}$ $W_{42}$ $W_{43}$ $W_{44}$
$C_{11}$ $C_{12}$ $C_{13}$ $C_{14}$
$C_{21}$ $C_{22}$ $C_{23}$ $C_{24}$
$C_{31}$ $C_{32}$ $C_{33}$ $C_{34}$
$C_{41}$ $C_{42}$ $C_{43}$ $C_{44}$
$A_1$
$A_2$
$A_3$
$A_4$

Figure 4.4: Schematic of MAC operation

Quantization maps floating-point values to an integer grid defined by parameters such as bit-width ($b$), scale factor ($s$), and zero-point ($z$, for asymmetric quantization). The scale factor specifies the step size, while the zero-point ensures that real zero is quantized without error.

**Symmetric unsigned quantization** For symmetric unsigned quantization (range $[0, 255]$), the step size is $s$. A number is represented as $s \cdot x_{\text{uint8}}$, where $x_{\text{uint8}}$ is an unsigned integer. Quantization is performed as:

$$x_{\text{int}} = \text{clamp}\left(\left\lfloor \frac{x}{s} \right\rceil ; 0, 2^b - 1\right)$$

And de-quantization is:

$$\hat{x} = s \cdot x_{\text{int}}$$

**Symmetric signed quantization** For symmetric signed quantization (range: $[-128, 127]$), the step size is $s$. A number is represented as $s \cdot x_{\text{int8}}$, where $x_{\text{int8}}$ is an unsigned integer. Quantization is:

$$x_{\text{int}} = \text{clamp}\left(\left\lfloor \frac{x}{s} \right\rceil ; -2^b, 2^b - 1\right)$$

And de-quantization remains:

$$\hat{x} = s \cdot x_{\text{int}}$$

**Asymmetric unsigned quantization** For asymmetric quantization (range: $[-sz, 255]$), the step size is $s$, and the representation is $s \cdot (x_{\text{int8}} - z)$. Quantization is:

$$x_{\text{int}} = \text{clamp}\left(\left\lfloor \frac{x}{s} + z \right\rceil ; 0, 2^b - 1\right)$$

And de-quantization is:

$$\hat{x} = s(x_{\text{int}} - z)$$

The quantization limits are:

$$q_{\min} = -sz \qquad q_{\max} = s(2^b - 1 - z)$$

Values outside this range are clipped, introducing clipping errors. Increasing the scale factor $s$ expands the quantization range, reducing clipping errors but increasing rounding errors, which are bounded by:

$$\left[-\frac{1}{2}s, \frac{1}{2}s\right]$$

This trade-off between clipping and rounding errors must be carefully managed to achieve optimal performance.

**Multiply and accumulate** A floating-point vector $\mathbf{x}$ can be approximated as a scalar multiplied by a vector of integer values. By quantizing the weights and activations, the quantized version of the MAC operation is expressed as:

$$\hat{A}_n = \hat{\mathbf{b}}_n + s_{\mathbf{w}} s_{\mathbf{x}} \sum_m \mathbf{W}_{n,m}^{\text{int}} \mathbf{x}_m^{\text{int}}$$

Here, $s_{\mathbf{w}}$ and $s_{\mathbf{x}}$ are scale factors for weights and activations, respectively. This formulation separates the scale factors, enabling efficient computation while maintaining precision.

#### 4.4.1.2 Usage

Quantization reduces the computational cost and memory footprint of NNs by converting floating-point representations into fixed-point representations. Two primary approaches exist: Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT).

**Post-Training Quantization** In PTQ, a pre-trained floating-point (FP32) network is directly converted into a fixed-point network without requiring retraining or access to the original training pipeline. The main challenge lies in determining the appropriate quantization range settings. Common methods include MinMax quantization and Mean Squared Error (MSE) optimization. MinMax quantization sets the range based on the minimum and maximum values of the tensor:

$$q_{\min} = \min \mathbf{V} \qquad q_{\max} = \max \mathbf{V}$$

MSE optimization minimizes the difference between the original and quantized tensors:

$$\underset{q_{\min}, q_{\max}}{\text{argmin}} = \left\| \mathbf{V} - \hat{\mathbf{V}}(q_{\min}, q_{\max}) \right\|_F^2$$

PTQ is effective and fast to implement but struggles with low-bit quantization, where large quantization errors may degrade performance.

**Quantization-Aware Training** QAT integrates quantization into the training process by simulating its effects during backpropagation. While this approach achieves higher accuracy, it requires longer training times and labeled data. A key challenge is handling the non-differentiability introduced by quantization. This is addressed using the straight-through estimator (STE), which approximates the gradient of the rounding operator as 1:

$$\frac{\partial \lfloor y \rceil}{\partial y} = 1$$

During the forward pass, quantization is applied, while the backward pass uses un-quantized values for gradient computation. QAT generally outperforms PTQ, especially for low-bit quantization, but at the cost of increased complexity.

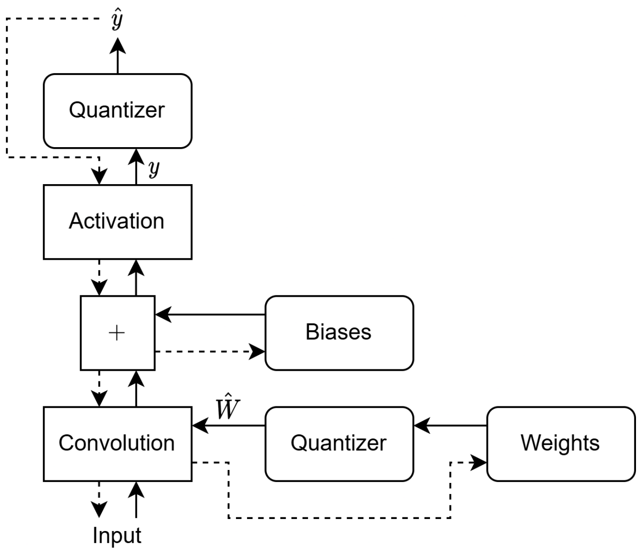

Figure 4.5: QAT forward and backward propagation

It is important to note that the best quantized network is not necessarily obtained by simply quantizing the best floating point network. Careful optimization is required to balance accuracy and efficiency.

### 4.4.2 Task dropping

Reducing the number of operations and model size improves computational efficiency and memory usage. Several techniques achieve this goal, including:

- *Network pruning*: NNs are often over-parameterized, meaning many weights are redundant and can be pruned (set to zero) without significantly affecting performance. Simple pruning removes individual weights, while structured pruning removes groups of weights, such as entire rows, columns, or filters. Aggressive pruning may require fine-tuning to recover accuracy.

- *Network architecture design*: efficient architectures replace large filters with smaller ones, reducing the total number of weights. Smaller filters can be concatenated before training to emulate larger ones, or tensor decomposition techniques can be applied after training to decompose filters without impacting accuracy.

- *Transfer learning*: transfer learning leverages pre-trained models by retaining the first few layers, which capture general features, and replacing the later layers with a simpler architecture tailored to the target task. This approach reduces the need for extensive retraining.

- *Knowledge distillation*: a smaller student network learns to mimic the behavior of a larger teacher network. The loss function minimizes the difference between the outputs of the two models, enabling the student to approximate the teacher's performance with fewer parameters.

## 4.5 Early Exit Neural Networks

Early Exit Neural Networks (EENNs) are designed to provide predictions within a fixed, often reduced, inference time. These networks allow predictions to be made at intermediate layers, reducing the computational load and energy consumption when full-depth inference is unnecessary. However, this improvement in computational efficiency comes at the cost of increased memory usage, due to the added classifier layers required at various depths of the network.

Traditionally, NNs are structured as sequential stacks of layers, with predictions generated only after all layers have been processed. While this ensures maximal feature extraction, it introduces several limitations: high computational demand, fixed and potentially long inference time, and risk of overfitting and other inefficiencies.

EENNs address these issues by incrementally processing inputs and making early predictions once sufficient confidence is achieved. This is enabled by Early Exit Classifiers (EECs) which are auxiliary classifiers inserted at intermediate stages of the network. The original model, without these exits, is referred to as the Backbone Network. EENNs leverage the observation that:

- Many input samples can be accurately classified with shallow sub-networks.

- The Lottery Ticket Hypothesis suggests that smaller, well-initialized sub-networks can match the performance of deeper ones.

- Overthinking can degrade performance, where deeper layers can override correct early predictions.

By allowing some inputs to exit early, EENNs not only reduce inference time but may also improve accuracy in certain scenarios.

EENNs bring several advantages such as reduced inference time, less overfitting, and mitigation of vanishing gradients and overthinking. Notably, earlier classifiers are not necessarily less accurate than deeper one.

**Classifier** Let $f_i(x)$ denote the output of an intermediate layer $i$ in the backbone network. An EEC is added at that point to produce a prediction:

$$\bar{y}_i = C_i(f_i(x))$$

Thus, the EENNs yields a sequence of predictions $\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_N$, in addition to the final output $\hat{y}$. These predictions may vary in accuracy depending on the layer at which they are computed.

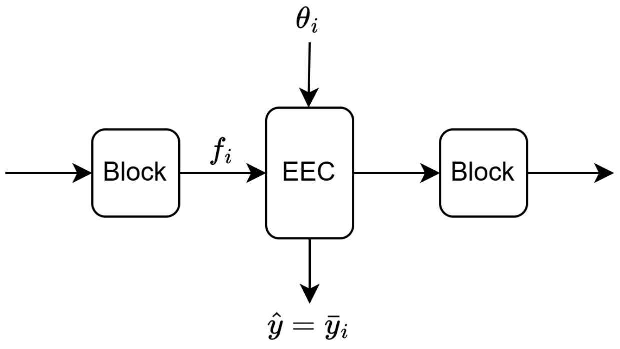

Figure 4.6: Early Exit Classifier

**Selection scheme** To determine whether an input should exit early, each EEC is associated with a decision function governed by a threshold. The decision function $D_i(x)$ compares the confidence score $C_i(x)$ against a predefined threshold $\theta_i$:

$$D_i(x) = C_i(x) \geq \theta_i$$

If the confidence score exceeds the threshold, the model halts further processing and outputs $\bar{y}_i$ as the final prediction. Otherwise, the input continues to propagate through the network. These thresholds $\theta_i$ are key hyperparameters that control the trade-off between inference time and classification accuracy.

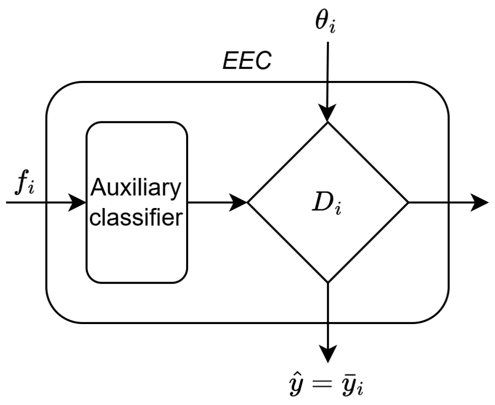

Figure 4.7: Early Exit Auxiliary Classifier and Decision Function

Common decision functions include:

1. *Maximum softmax probability*: let $S_i(x)$ be the softmax output of EEC$_i$ for input $x$. The confidence is given by:

$$\text{confidence}(x) = \max S_i(x)$$

   Here, $S_i^j(x)$ is the softmax probability for class $j$.

2. *Score margin*: defined as the difference between the top two softmax scores:

$$\text{SM}_i = S_i^{(1)}(x) - S_i^{(2)}(x)$$

   Here, $S_i^{(1)}(x)$ and $S_i^{(2)}(x)$ are the highest and second-highest probabilities, respectively. A larger margin indicates higher confidence.

3. *Entropy*: measures uncertainty in the prediction:

$$H(y) = -\sum_{i=0}^{C} y_i \log(y_i)$$

Here, $C$ is the number of classes. Entropy is minimal when the output is a one-hot vector and maximal for a uniform distribution.

### 4.5.1 Training

The training of EENNs typically falls into three main strategies:

1. *Joint training*: in this approach, all EECs are trained simultaneously by optimizing a combined loss function:

$$\mathcal{L}_{\text{joint}} = \mathcal{L}(\hat{y}, y) + \sum_{i=1}^{N} w_i \mathcal{L}(\bar{y}_i, y)$$

   Here, $\mathcal{L}$ is the standard cross-entropy loss, $\hat{y}$ is the output of the final classifier, $\bar{y}_i$ is the output of the $i$-th EEC, $y$ is the ground truth label, $w_i$ are weighting factors, which can be uniform or treated as tunable hyperparameters, and $N$ is the total number of EECs.

2. *Layer-wise training*: this strategy trains the network progressively. At each stage, a new EEC and the corresponding portion of the backbone are trained while keeping the previously trained layers frozen. This approach allows the network to stabilize lower layers before deeper blocks are optimized.

3. *Knowledge distillation*: in this two-stage method, the backbone network (acting as the teacher) is trained first. Once trained, the EECs (students) are trained on top of the frozen backbone, learning from both the ground truth and optionally from the teacher's soft predictions.

### 4.5.2 Inference

Two primary inference strategies can be adopted for EENNs:

- *Full evaluation*: the input is propagated through the entire network, and predictions from all EECs are aggregated to form a final decision.

- *Early-exit inference*: the network sequentially evaluates each EEC and stops computation as soon as a confident prediction is made. This enables computational savings by avoiding unnecessary processing of deeper layers.

### 4.5.3 Architecture

**BranchyNet** BranchyNet is built on top of the original AlexNet architecture, adding two EECs at intermediate layers of the network. The model is trained using a joint training strategy. During inference, computation halts immediately once a sample meets the exit condition at an EEC, avoiding unnecessary forward propagation through deeper layers. The decision to exit is based on a confidence measure computed from the entropy.

**Gate-classification** Gate-classification, on the other hand, take a probabilistic approach to confidence estimation. In these models, confidence is explicitly modeled as the posterior probability of the predicted class given the input. Inference proceeds sequentially through the network, and the decision to exit is made as soon as this posterior probability exceeds a predefined threshold.

CHAPTER 5

# On-device learning

## 5.1 Introduction

We want to address the level six of TinyML in which we have both training and inference entirely on end device. We want to adapt at various levels:

1. *Environmental*: we may have to analyze data coming from different situations and weather so the data may change. Therefore, we need to adapt the data acquisition and preprocessing based on the actual state of the environment.
2. *System*: the system itself may operate under different hardware constraints, network conditions, or computational capabilities. Thus, it is essential to adapt resource allocation, communication protocols, and processing strategies to ensure robustness, efficiency, and real-time performance across heterogeneous setups.
3. *Task*: different tasks may require different models, processing pipelines, or decision-making criteria. Adapting to the task means selecting the appropriate algorithms, models, or data representations dynamically.
4. *User*: users may have different preferences, goals, or behaviors. Therefore, the system should adapt.

The problem with adaptation is that the system may change its behavior or not function properly.

The most significant technological challenge lies in computational constraints, particularly memory limitations (for storing data and model weights in RAM). Training is considerably more demanding than inference in terms of computational time. Since the new data lacks labels, an alternative approach is needed to classify the incoming data samples. Validation also becomes more complex, as it is increasingly difficult to compare different solutions effectively.

## 5.2 Inference and training

In the inference phase, we use a test set $\{x_t\}$ and a TinyML model that processes these input samples to produce outputs relevant to the task:

$$y_t = f_\theta(x_t)$$

Here, $f_\theta(\cdot)$ is the model parameterized by weights $\theta$, and $y_t$ is the predicted output at time $t$.

To enable on-device training, we require a labeled dataset collected on the device, referred to as the on-device training set:

$$O_t = \{(x_M, y_M)\}$$

We also need a learning algorithm equipped with a loss function to update the model parameters. The new weights at time $t$ are learned as:

$$\theta_t = \mathcal{L}(f_{\theta_{t-1}}(\cdot), O_t)$$

Here, $\mathcal{L}$ represents the learning algorithm applied to the model and the current on-device training data.

Beyond standard learning, a meta-learning algorithm is needed to select the most appropriate model architecture or configuration for the task. This process identifies the function $f_\theta(\cdot)$ from a class of models, based on the training data, improving adaptability across tasks or environments.

We also require a monitoring algorithm, which evaluates the relevance or utility of training data and model behavior. This can be described as:

$$a_t = \mathrm{CdT}(O_t, f_\theta(x_t))$$

Here, CdT stands for change detection and triggering, a mechanism that monitors performance or data distribution shifts and determines whether model adaptation or retraining is necessary.

Compared to the baseline case model inference, this setup introduces continuous on-device training, which runs in parallel with inference. This enables the system to adapt over time as new data becomes available, improving robustness and personalization.

### 5.2.1 On-device inference

In the context of on-device inference, the behavior of the model $f_\theta(\cdot)$ is fixed and does not change after deployment. The primary objective in this scenario is to minimize both the memory footprint and the computational demand.

Traditional solutions developed at design time include techniques such as fixed or variable quantization to reduce the precision of weights and activations, structured and unstructured pruning to eliminate redundant parameters, and knowledge distillation to transfer information from a large model to a smaller, more efficient one.

More recent approaches involve run-time adaptability. These include adaptive quantization, where the precision is adjusted based on the input or system constraints, and early-exit NNs, which can terminate computation early and provide intermediate outputs when the confidence of predictions is sufficiently high. These strategies allow for dynamic optimization without compromising model correctness.

### 5.2.2 On-device personalization

On-device personalization focuses on optimizing task performance for a specific user, context, or environment, while remaining within the device's memory and computation constraints. This typically involves fine-tuning the model $f_\theta(\cdot)$ to produce a specialized version that better fits the personalized data characteristics.

A crucial requirement for this process is the availability of supervised information. One of the central challenges is ensuring that the adapted model actually improves over the original $f_\theta(\cdot)$, especially given the limited availability of validation data on the device. Without reliable validation, the risk of overfitting or negative adaptation remains significant.

### 5.2.3 On-device learning

On-device learning extends the concept of personalization by enabling continuous model adaptation over time. The main bottleneck in this paradigm is the memory cost of maintaining the on-device training set $O_t$, which may quickly consume available RAM or flash storage. Additionally, supervised on-device learning relies on access to ground-truth labels, which may not be consistently available in practical scenarios.

The adaptation process is governed by a monitoring algorithm that determines when model updates should be triggered. For example, change detection tests (CdT) can be used to identify shifts in data distribution or prediction confidence, thereby signaling the need for retraining. This monitoring can be active, reacting dynamically to changes, or passive, following predefined intervals or triggers.

The way in which $f_{\theta_t}(x_t)$ evolves over time is defined by a meta-learning algorithm. Depending on the design, this may follow fixed rules, such as periodic updates, or more flexible schemes that adapt based on performance or uncertainty estimates. Due to the constrained environment, explicit model selection is typically not feasible, and models must adapt within predefined structural bounds.

Finally, the adaptation algorithm defines what part of the model should be updated, such as the last layers in a CNN. The lack of a validation phase poses a challenge to ensuring that the adapted model maintains or improves performance. Guaranteeing the correctness and safety of these adaptations is essential, particularly in safety-critical or user-facing applications.

**CHAPTER 6**

# Artificial Intelligence ethics

## 6.1 Datafication

Data plays a fundamental role in enabling current forms of AI, including applications at the edge. However, the acquisition and use of data come with significant challenges. These include scarcity, high costs, and ethical or regulatory limitations. In addition, growing concerns about the energy consumption and environmental impact of data-centric technologies have been raised.

We are witnessing an unprecedented expansion in the presence and production of data (a trend driven by the increasing importance of datafication).

**Definition** (*Datafication*)**.** Datafication refers to the process of transforming objects, behaviors, and activities into digital data.

This is not a purely technical operation but one that is embedded in broader social, economic, and political contexts. For datafication to occur, several key conditions and actors must be present:

1. *Community*: a network of users, professionals, researchers, hackers, and innovators who interact with and contribute to the data ecosystem.
2. *Care*: the values, incentives, practices, and regulatory standards that ensure the responsible handling of data and maintain trust in data-driven systems.
3. *Capacities*: the technical infrastructure required for data production and processing, including sensors, platforms, cloud computing, storage, and analytical tools.
4. *Data*: the actual outputs of datafication which includes digital traces, sensor measurements, content, metadata, and more.

Importantly, data do not exist or carry meaning on their own. Their utility and legitimacy depend on the interplay between these four elements.

Thus, datafication is not merely a technological process; it is deeply social and political. It requires alignment between infrastructure, stakeholders, and norms. When these elements are not in sync—when there is resistance, lack of care, or misalignment of interests—the processes of datafication can falter or generate conflict.

## 6.2 Classification

AI systems, particularly those deployed at the edge, function fundamentally as classification machines. These systems are designed to categorize and interpret input data to perform tasks such as activity recognition, automated measurement and analysis, and generating recommendations and indications.

At the core of these capabilities lies the act of classification. As Bowker and Star (2000) argue, classification is not merely a technical activity but a foundational operation in scientific inquiry and social organization. It involves:

- The development of categories and kinds.
- Structuring the ways we understand, predict, and explain the world.
- Enabling inference, explanation, and decision-making.

However, classification is not a neutral or purely objective process. It carries with it the power to naturalize particular views of the world. That is:

- Categories created by AI systems are not necessarily natural or inevitable.
- To classify phenomena is to reify the categories.
- Once institutionalized, these categories can stabilize.

This means that data and classification schemes actively shape the world rather than merely describing it. They define what is seen, what is ignored, and how meaning is constructed.

Classifying people or behaviors can reinforce specific interpretations and value systems. These classifications include certain identities or actions while excluding others. Over time, they not only assign meaning to entities but also lend authority to the categories themselves. Thus, AI systems do not just use data; they participate in the making of social and material realities.

## 6.3 Neutrality

While AI is often discussed in technical terms, it is shaped by a series of social, ethical, and political decisions. From the initial stages of data collection to the implementation of classification mechanisms, each step in the development and deployment of AI systems involves value-laden choices.

AI systems are not merely computational tools. They are social-technical systems (Johnson and Verdicchio, 2017), shaped by the interplay between technological artifacts, human behavior, institutional arrangements, and social meanings.

These systems embody several important choices:

- How care is delivered and organized.
- How health systems integrate AI.
- Whether medicine becomes increasingly personalized.
- How data are treated (as assets, liabilities, or public goods).

However, these systems also raise concerns:

- High computational demands for model training and maintenance.
- Simplifications in measurement.
- Tensions between technical feasibility and clinical or contextual relevance.
- Questions of professional judgment, accountability, and ethical oversight.

These considerations challenge the notion that AI systems are neutral or purely objective. As Johnson and Verdicchio (2017) argue, AI systems should be understood as embedded within complex social-technical environments:

- They embody political, philosophical, and technical choices.
- They consist of interactions between artifacts, human users, and institutional settings.
- Their function depends not only on technical accuracy but also on meaning.

In short, AI systems are never value-free. They are deeply shaped by and in turn help shape. Recognizing this non-neutrality is essential for responsible development, deployment, and governance of AI, especially in sensitive areas such as health, education, and public infrastructure.