Notes on Hardware Architectures for Artificial Intelligence

Mattia Morabito

2023/25/02

Contents

1 License

2 Introduction

Before beginning any discussion on the topics of the course it's better to keep in mind the definition of Artificial Intelligence to which we will refer during the duration of the course:

Artificial Intelligence is the area of computer science that studies the development of hardware and software systems endowed with abilities typical of human beings. Such systems are able to autonomously pursue a given purpose by making decisions that, until then, were usually made by human beings.

In particular by hardware and software systems we intend to describe a particular stack composed by:

- **AI Software (Application)**: The AI-based application running into the IT system
- **AI Software (Framework/Platform/Tool)**: Programs and libraries to control the physical resources and provide tools to build applications (i.e. TensorFlow)
- **AI Hardware**: Hardware to support AI, either specialized or general purpose

2.1 AI Hardware classification

In the realm of AI Hardware we can list everything from full-on datacenters to tiny IoT nodes. This broad classification while helping us understand the reach AI in our modern day does not help as designers. How our application should run? Should it be optimized for data centers? What are the advantages and disadvantages of each approach?

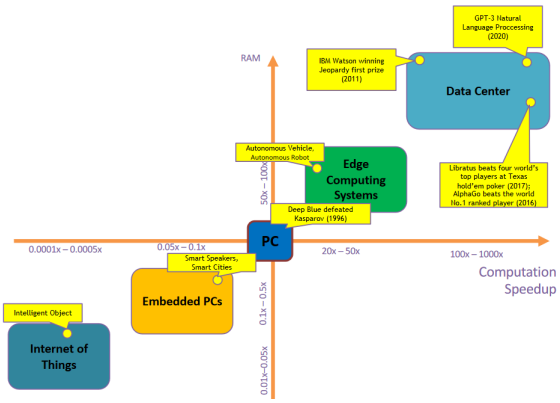

Figure 1: AI Hardware classification

Let's split the spectrum in 4 main sections:

2.1.1 Data Centers

The datacenter is the perfect environment to run scalable and heavy applications due to the basically infinite amount of processing and storage resources. In this way we can implement high performance models while lowering our IT cost and maximizing data reliability, however everything comes at a cost, in particular we require a constant, stable and high speed network connection. Moreover, we

should particularly focus on security and privacy due to possible attacks while considering a very high power consumption due to constant transmission of data.

Datacenters are also not able to fit the requirements of time-critical applications such as autonomous driving due to unpredictable latency of the transmission. This single drawback rules them out of consideration and poses a need for more responsive and predictable systems in terms of latency.

2.1.2 Edge nodes

Due to the previous flaws we must move the computation from the cloud to the real world. A fist approach is to move some servers near to the application as is the case for high-speed networking (5G). This approach is commonly referred as *Edge Computing* where some Edge nodes are directly integrated in the field. This approach brings high computational capacity, reduced latency for time-critical applications and increases the security of the system (decoupling networks and users).

However, edge nodes are still powerful machines and require a constant power connection while also being connected to cloud.

The term *Edge* is usually in reference to the computational activity, while for networking we use the term *Fog*

2.1.3 Embedded computers

Embedded PC go a step further, they reduce performance while still providing a pretty competent machine while allowing to be distributed in the environment. One of the advantages is the availability of many development board which are ready to be programmed while also providing a standard OS interface which hugely simplify the development of the application.

Still they can be pretty demanding in terms of power, so they need a power connection, and require some hardware design.

2.1.4 IoT devices

IoT devices are the most integrated and distributed components of the infrastructure. In fact by transition from microprocessors to microcontroller we can hugely reduce their power consumption ($\simeq mW$) allowing them to be battery powered. Cost also goes down while still allowing to allocate sensors and actuators to gather data and interact with the environment.

The processing power is greatly reduced, posing the need for more involved software (i.e. lack of OS interface) and hardware design (i.e. tight memory and energy requirements) to get the most out of each device. By forgoing the cabled connection we also introduce additional challenges related to wireless connectivity (synchronization, networking).

We should also clarify that there is a difference between a *Connected object*, which is able to connect to the network and exchange data, and a *Smart Object* which is also connected but is also able to make decisions based on the information that it gathers.

2.2 Objectives

Our objective is to *move (intelligent) processing as close as possible to data generation units.* This brings a lot of advantages such as:

- **Increasing autonomy**: by introducing more smart objects which do not rely on other systems
- **Reducing decision-making latency**: fewer unpredictable latencies
- **Reducing transmission bandwidth**: fewer cloud-node communication
- **Increasing energy-efficiency**: since computing is $\simeq 10x$ more efficient than transmitting
- **Increasing Security and Privacy**: local data is processed on the spot and not shared with the whole network
- **Incremental/Adaptive Learning**: environment impervious systems to correctly scale performance
- **Exploiting the ecosystem of units**:

This however poses some rather interesting challenges due to:

- **Hardware constraints**: which includes low computational ability, energy and memory
- **Complexity in design and development**: due to the lack of OS interfaces
- **Strong connection between HW, SW and ML**: poising the problem of tight integration

3 Embedded and Edge Hardware

Embedded and edge hardware is a very diverse and heterogeneous landscape, however we should find some common architecture to better analyze our requirements and our possible solutions. First let's clarify that for *Embedded systems* we mean computers that are designed to perform one specific task (Not general purpose).

In particular *Embedded system* are meant to rum a specific software referred as *Embedded software*.

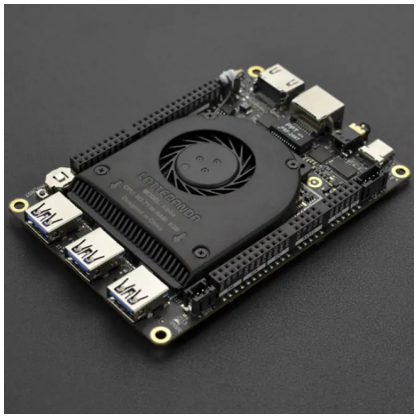

Figure 2: Lattepanda SBC (Single Board Computer)

3.1 Architecture

A generic *Edge device* or *Embedded system* will have the following architecture, as described in the following figure:

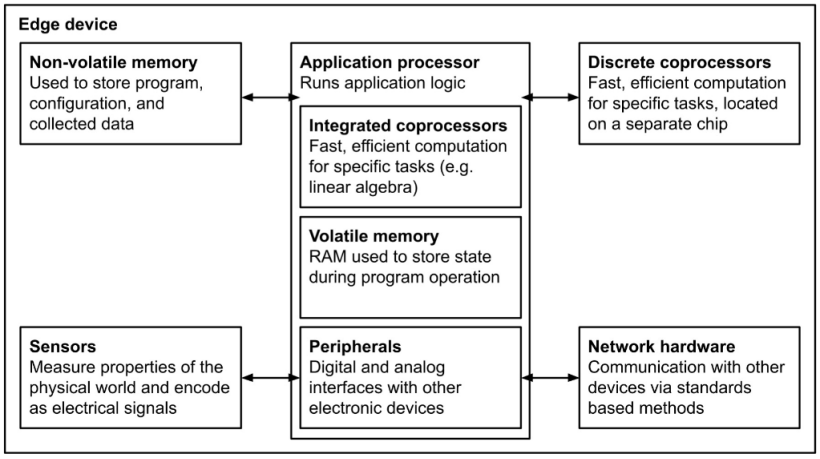

Figure 3: Architecture of an edge device

3.1.1 Application processor

The application processor is a General Purpose Processor (GPP) that coordinates the application and runs most (if not all) the algorithms and logic that make up its program. This class of processor in many cases comes pre-equipped with *integrated coprocessor* (or tightly coupled accelerators), built-in hardware to efficiently perform some specific computation (Cryptographic algorithms, Floating point arithmetic, DSP etc.).

In the same package the GPP integrates *Volatile memory* used as working memory during the program execution, however this does not preclude the system from having additional memory outside the package. RAM is usually the bottleneck for Edge devices due to the high area and power requirements. Peripherals are also bundled in the same package to enable communication between external electronics devices and the processor.

3.1.2 Non-Volatile memory

Non-volatile memory are storage devices which are able to maintain data after power off. This is crucial to retain information such as software programs, user configuration, and machine learning models. It's slow to read and extremely slow to write. Even if in the diagram this component is displayed as off-chip in practice some small amount of EEPROM or NAND-Flash can be implemented directly in the GPP chip.

This kind of memory is absolutely necessary if we think that some application may require our system to be powered on only for a few second every hour (Sampling of some sensors every hour).

3.1.3 Discrete coprocessors

Discrete coprocessor are basically the same as integrated coprocessor but are located off-chip. This allows for much more complex systems where the coprocessor may be even more powerful (and power hungry) than the application processor (i.e. Ultra-Low-Power SoC with a GPU).

3.1.4 Sensors

Sensors are the interface that our electronic system has with the world, allowing him to gather measurements from the environment and/or from humans. Sensor can be *physical*, as in the measure some physical quantity, or they can be *virtual*, measuring some implicit quantity such as transmission speed of a communication channel.

As output, we get streams of data which are provided in different data formats.

3.2 Data storage

Now that we understand how our system is composed we should investigate how to gather and store data for our application. In particular each data value can be stored as data structure, such as *Boolean*, *n-bit integers (or Fixed-Point)* or as *n-bit Floating point*.

> Storing data as a floating point may be convenient but keep in mind that an operation usually requires $\simeq 10 \div 20$ times more energy than the Fixed-Point counterpart

Various sensors also organize data differently such as:

3.2.1 Time Series

A time series is a sequence of data points indexed in chronological order, at a fixed sampling rate or period (equally spaced). The memory requirement can be computed knowing the sampling period T_s and the number of bits n used for the representation as:

$$N[bits] = \frac{T}{T_s} * n \tag{1}$$

Where T is the amount of time for which we sample data

3.2.2 Audio

A special case of time series data, audio signals represent the oscillation of sound waves as they travel through the air. The *Sampling frequency* $f_s[Hz]$, *Quantization or bits per sample* $n[bits]$, *length of the signal* $T[s]$ and *number of channels* C (e.g., mono or stereo) are key aspects. Once all the parameters are known we can compute the memory requirement as:

$$N[bits] = T * f_s * C * n \tag{2}$$

3.2.3 Images

Images are data that represent the measurements taken by a sensor that captures an entire scene. They have two or more dimensions. In their typical form, they can be thought of as a grid of "pixels", where the value of each pixel represents some property of the scene at the corresponding point in space (stored in n bits). Assuming an image has dimension $W \times K$ with C channels we can compute the memory requirements as:

$$N[bits] = W * K * C * n \tag{3}$$

3.2.4 Videos

A sequence of (fixed) images reproduced with a suitably high frame rate Similarly to image we have W, K, n and C (channels). In addition, we have a *frame rate* $F_r[frames/s]$ and the *length of the video* $T[s]$. Once all the parameters are known we can compute the memory requirement as:

$$N[bits] = T * F_r * W * K * C * n \tag{4}$$

This computation is valid for *raw or uncompressed* video

3.3 Sensors

To give a more complete picture of the embedded hardware we will now list some of the most relevant families of sensor on the market

3.3.1 Acoustic and vibration

Acoustic and vibration sensor allow us to detect the effects of movement, vibration, and human and animal communication at a distance by measuring the effect of vibrations travelling through a medium like air (microphones), water (hydrophones) or ground (geophones and seismometers).

This family of sensors generally produce an audio data describing the variation of pressure in the medium. This information is distributed across frequencies, making the sampling rate (or frequency) the most important parameter (Nyquist).

3.3.2 Visual and scene

Camera allow us to acquire the scene information without contact. They can range from tiny and low-power to high quality multi-megapixel sensor They are generally characterized based on *Color channels*, *spectral response* (IR, X-RAY), *pixel size* (sensitivity), *resolution* and *frame rate*. Their output is an image (2D/3D) or a video.

3.3.3 Motion and position

Motion and position sensors allow us to obtain the spatial information of a target such as position, velocity or acceleration. They come in many types and configurations but the most prominent are:

- **Tilt sensors**: which measure orientation by a mechanical switch.
- **Accelerometers**
- **Gyroscopes**: which measure the rate of rotation;
- **Time of flight (ToF)**: which measure the distance of the target object from the sensor by electromagnetic emissions.
- **Real time locating systems**: multiple transceivers in fixed locations around a building to track the position of individual objects.
- **GPS**: which use satellites to determine the location of a device.

Their measurements are usually represented as time series.

3.3.4 Force and tactile

Helpful in facilitating user interaction, understanding the flow of liquids and gases, or measuring the mechanical strain on an object:

- **Buttons and switches**: provide a binary signal.
- **Capacitative touch sensors**: measure the amount that a surface is being touched by a conductive object, like a human finger.
- **Strain gauges**: measure how much an object is being deformed.
- **Load cells**: these measure the amount of physical load that is applied.
- **Flow sensors**: measure the rate of flow in liquids and gases.
- **Pressure sensors**: used to measure pressure of a gas or liquid, either environmental or inside a system (e.g., inside a car tire).

Measurements are typically represented as a time series.

3.3.5 Optical, electromagnetic and radiation

Sensors measuring electromagnetic radiation, magnetic fields as well as current and voltage:

- **Photosensors**: detect light at various wavelengths, both visible and invisible to the human eye.
- **Color sensor**: photosensors to measure the precise color of a surface, helpful for recognizing different types of objects.
- **Spectroscopy sensors**: measure the way that various wavelengths of light are absorbed and reflected by materials, giving an edge AI system insight into their composition.
- **Magnetometer**: measure the strength and direction of magnetic fields (a digital compass).
- **Inductive proximity sensors**: electromagnetic field to detect nearby metal (detect vehicles for traffic monitoring).
- **Electromagnetic field (EMF) meters**: measure the strength of electromagnetic fields (e.g., emitted by industrial equipment).
- **Current and voltage sensor**

Measurements are typically represented as a time series.

3.3.6 Environmental and chemical

Sensors measuring physical quantities related to the environment in which the device is in. The more common one include:

- **Temperature sensors**
- **Gas sensors**: e.g., humidity or carbon dioxide sensors.
- **Particulate matter sensor**: monitor pollution levels.
- **Biosignals sensors**: e.g., measurement of electrical activity in the human heart (electrocardiography) and brain (electroencephalography).
- **Chemical sensors**: measure the presence or concentration of specific chemicals.

Measurements are typically represented as a time series

3.4 Our choice

In this course we are going to deal mostly with Microcontrollers which will represent the basic technological block of our pervasive applications. Their size, generally low cost the fact that every useful component is already embedded in the same piece of silicon makes them the perfect choice.

However, we must also take into consideration that there is no operating system on this platform, so the *(*firmware) is directly executed on the hardware and includes the low-level instructions to connect the peripherals. The programming language can either be high-level (C, C++) or low-level (platform specific Assembly).

Other platform like SoC are present in the embedded AI landscape but for out purpose the higher computing power is not worth the low energy efficiency (w.r.t. MCUs) and the larger cost. By keeping in mind the data storage and computation requirements we can now compile a summary table representing the capabilities of each platform.

Device vs Data		Time-Series		Audio	Images		Video
		Low-frequency	High-frequency		Low resolution	High resolution	
MCU	Low-end						
	High-end						
	High-end with accelerator						
SoC	Vanilla						
	With accelerator						
Edge Server							
Cloud							

Table 1: Summary of platform capabilities

4 Algorithms for Embedded and Edge AI

Now that we have a clear view of the hardware architecture for our system we can begin discussing the AI Application that it will be running The software can now be segmented in 3 main components as shown in figure:

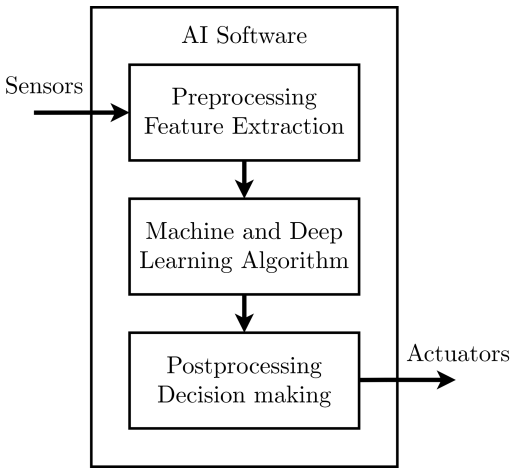

Figure 4: AI Software stack

4.1 Preprocessing & Feature Extraction

The output of our sensor is a raw and noisy data stream which must be processed before entering our Machine/Deep Learning Model. This goal can be achieved by 3 main methodologies.

4.1.1 Data segmentation

The stream of data coming into our system is continuous however the model can only work on a finite set of data, for this reason we must find a method to correctly segment our stream into blocks which will then be used as input in the algorithm. The length of this data block is defined by us and must take into consideration the type of data, the *latency* of the algorithm and the memory constraints.

This before mentioned *latency* defines how fast our system can process a chunk of data. While this constraint is usually neglected in cloud application, were the inference time is always negligible or not of interest, in embedded systems which follow the *always on Machine Learning* approach is a key parameter. A simple example can be made considering autonomous driving vehicles, where the algorithm governing the steering and braking must be able to react in fractions of second to avoid accidents. The design choice of the algorithm has a strong effect on the computational requirement (the latency) as well as the memory demand.

So by expanding our data chunk we increase the amount of information reaching our model, in turn increasing the accuracy, but both our memory requirements and latency increase which may or may not be acceptable depending on the application.

Even if we decouple data acquisition and processing latency will still set our system maximum *frame rate* (the rate at which the system is able to acquire and process the data). The limited frame

rate also creates the problem of data loss, while our system is processing we can't store an infinite amount of data, and some of the chunk that we discard are bound to have relevant information in them. This issue can be slightly mitigated if we overlap the data chunks, however while this decrease the risk of losing information it requires lower latency to make it work correctly and renders our observation non i.i.d (independent identically distributed).

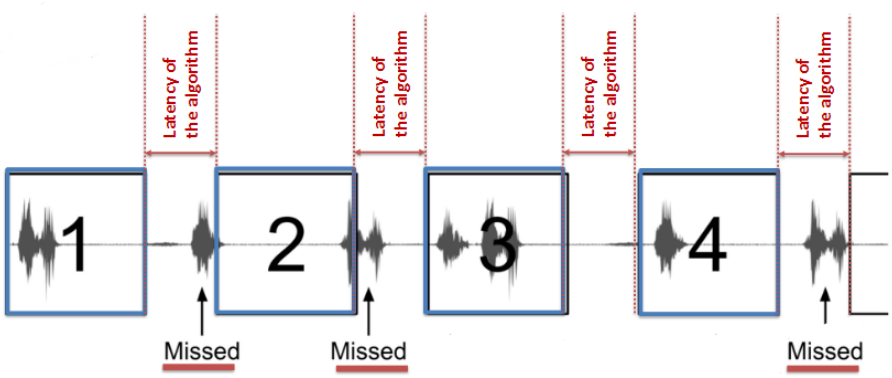

Figure 5: Example of missed data due to latency

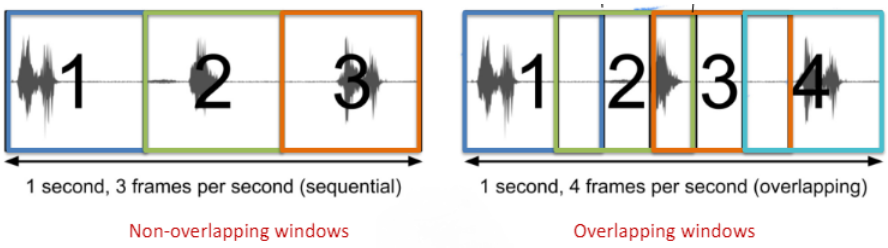

Figure 6: Chunk Overlapping

4.1.2 Digital processing algorithms for data preprocessing

Once our chunks are defined we can apply digital signal processing algorithms to remove noise and highlight relevant portions of the signal.

As a first step if some data points are missing we want to reconstruct them by either:

- **Global filling methods**: in which we use the whole chunk to fill the missing observations.
- **Local filling methods**: in which we use neighboring data to fill the missing observation (only applicable in situation where for each data point we should make a decision).
- **Deletion**: in which we choose not to use time periods that have missing data at all.

Also, we might have situations in which time series have different sampling frequencies. This is a problem for our model and is fixable by either:

- **Upsampling**: increasing the sampling frequency artificially by adding data points in between observations (replication, interpolation). We must be aware that while this is a viable method

we increase the memory footprint while not adding any relevant information.

- **Downsampling**: decreasing the sampling frequency artificially by only saving a data point every N (subsampling). In this method while we are losing significant information we are also reducing the memory footprint.

In general, we normalize to the slowest sample while minding the aliasing. This is also applicable to spacial information such as images but instead of frequency we deal with spacial resolution either by keeping the shape of the data (downsampling, interpolation) or by modifying it (cropping, resizing). It is particularly useful since usually the maximum input size of the model is a constraint given by memory limitation (model, data size).

Last but not least the data may need to filtered to only consider information relevant to the application. This is quite convenient since many embedded and processor support fast and energy-efficient computation of filtering operations (LP, HP, BP).

4.1.3 Feature Extraction

Extracting the relevant features in some data is a challenging task and usually requires the help from a field expert however there are some basic analysis techniques that we can employ such as:

Domain		
Time		Frequency
Mean	SNR	Variance of peaks
PCA	Peak Decay	Max amplitude
Max amplitude	Energy	Main frequencies
Spectrogram		

Table 2: Feature extraction basic techniques

Images also have some basics techniques for feature detection, mostly implemented by libraries such as *OpenCV* (for SoC) or *OpenMV* (for MCUs). Some examples are *Edge, Corner, Blob and Ridge* detection.

4.1.4 Feature combination and data normalization

As a last step me must combine all of our feature together to create the input vector of our model.

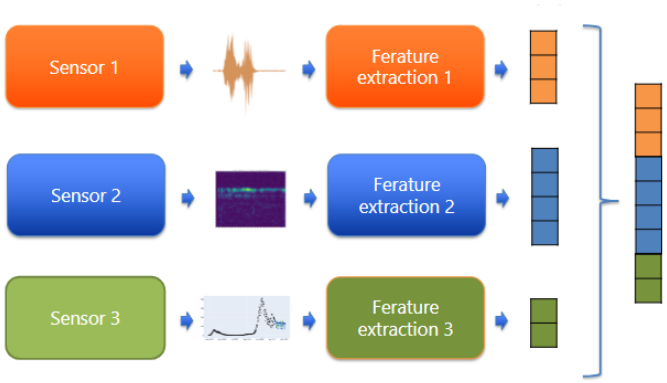

Figure 7: Feature combination example

While simple consideration have to made in case of features with different scales which may lead to model under/over fitting. This can be avoided by:

- **Min-Max Normalization**: normalizing the data to have everything in the $[0, 1]$ range

$$x_i^{norm} = \frac{x_i - MIN_i(x_i)}{MAX_i(x_i) - MIN_i(x_i)} \tag{5}$$

- **Standardization**: removing mean and dividing by the standard deviation

$$x_i^{std} = \frac{x_i - MEAN(x_i)}{STD(x_i)} \tag{6}$$

Where all parameter may vary with time and must be correctly tuned to achieve the tightest range to maximize the model performance.

4.2 Machine and Deep Learning Algorithm

Our goal now is to apply machine/deep learning algorithms on the preprocessed data to generate and output, however since the listing every algorithm will be impossible it is best to deliver a first order classification of algorithms based on their functionality:

Functionality	Examples	
Classification	Classify micro-acoustic emissions of a rock face	Recognize bird vocalizations
Regression	Measure the level of a liquid in cup	Estimate the number of people in the room
Object detection	OpenDataCam	Counting the number of attendees to an event
Segmentation	Identification of logically coherent regions of data	Street view for autonomous driving
Anomaly detection	Detect persons in a conveyor belt	Detect micro-acoustic emissions
Prediction	Predict the next value of a time series	Weather forecasting
Feature reduction	Compress or extract features from audio chunks to reduce the size	

Table 3: Algorithm's classification by functionality

Another possible classification can be achieved by considering the implementation, as such we may have:

- **Conditional logic**: In cases where we can find a deterministic model to carry out the decision we may employ a series of conditional statements to implement the application. This is not machine learning however since the system does not learn anything and is inflexible to more general cases. Does not require training and is the most efficient way to implement our application.
- **Machine Learning**: In machine learning before using the model we must rely on a field-specific expert to extract the relevant feature, which then become the input of our model. In other word the machine learning model only learns the how to classify based on the features.
- **Deep Learning**: In some situation we may not have the capabilities to extract the relevant features from out data, deep learning solves this issue by allowing the model to carry out this step. While this seems like the obvious choice this class of algorithm require much deeper pipelines and is rather data-hungry. Also, by decoupling the feature extraction we have no transparency on the decision policy (i.e. a model may learn to classify an image by the background and not the subject).

In this course we will focus on implementing machine and deep learning algorithms on IoT and Edge systems, in this context our framework of choice is *TinyML*. It is mandatory to point out that not the whole tool chain is necessarily implemented on the distributed devices. Some policy choices regarding this can be found on the following figure:

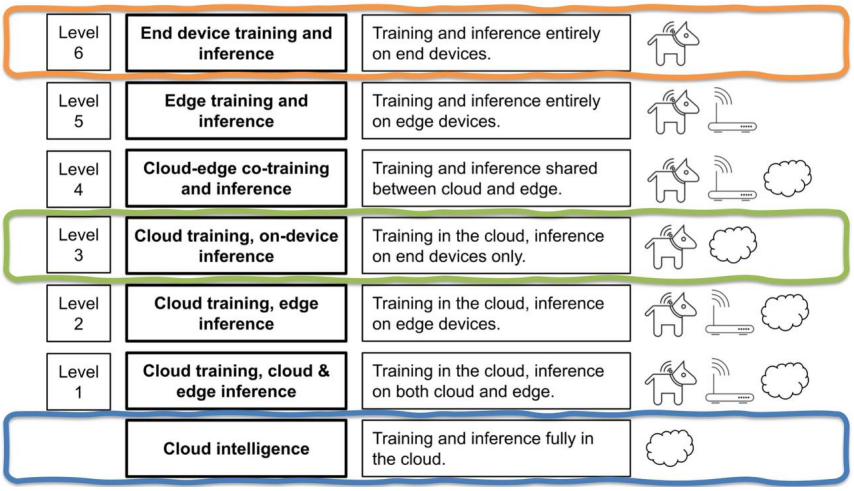

Figure 8: Levels of integrations concerning AI

4.3 Post-processing & Decision-making

5 Basics of Machine learning

It is important to review some basics concepts of machine learning to fully understand the future subjects. As such, in general, the goal of learning given a data generating process is to identify and build the simplest approximating model able to explain both the past and future instances of the process.

To this aim we must identify the possible sources of error (or risk) in our model. An elegant and complete representation of the error sources is:

$$\epsilon = \epsilon_E + \epsilon_A + \epsilon_I \tag{7}$$

Where the total error is the superposition of:

- ϵ_I is the error that depends only on the structure of the learning problem and, for this reason, can be reduced only by improving the problem itself (i.e. non-removable noise).
- ϵ_A is the approximation error that depends on how close the model family is to the process generating data (i.e. is the model the correct approximation of the process). In the tinyML space this is the most relevant contribution.
- ϵ_E is the estimation error that depends on the ability of the learning algorithm to select parameters which correctly represent the process. Mainly dependent on the amount of data available.

We will mostly discuss neural networks since the *Universal approximation theorem* states:

> A feedforward network with a single hidden layer containing a finite number of neurons approximates any continuous function defined on compact subsets.

Which, while not giving any constructive information about the number of neurons is actually required, allows us to assume that the approximation error will be always 0. As a consequence we need to worry only about the estimation error.

But how can we evaluate the performance of our model? Below are listed some common validation techniques such as:

- *Apparent Error Rate (AER), or resubstitution*: The whole set is used both to infer the model and to estimate its error
- *Sample Partitioning (SP)*: Training set and Evaluation set are obtained by randomly splitting the whole set in two disjoint subsets. The training set is used to estimate the model and the evaluation set to estimate its accuracy.
- *Leaving-One-Out (LOO)*: The evaluation set contains one pattern in the set, and the training set contains the remaining $n - 1$ patterns. The procedure is iterated n times by holding out each pattern in the whole set, and the resulting n estimates are averaged.
- *W-fold Crossvalidation (wCV)*: The set is randomly split into w disjoint subsets of equal size. For each subset the remaining $w - 1$ subsets are merged to form the training set and the reserved subset is used as evaluation set. The w estimates are averaged.

6 Model requirements

We now want to analyze the sources of complexity coming from our model, both in memory and computational terms. To this aim we can use as a case study the convolutional neural network in figure:

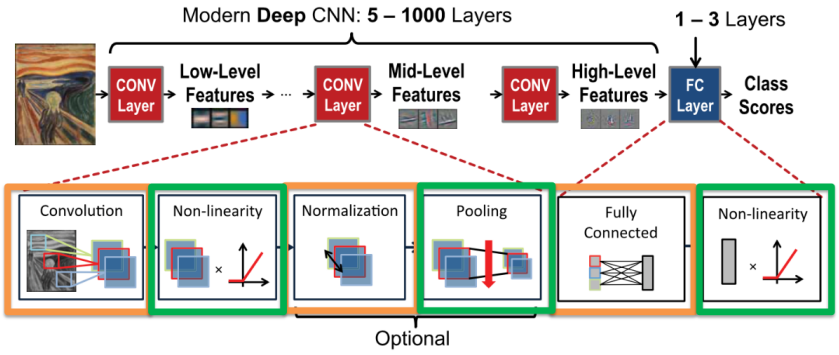

Figure 9: Convolutional Neural Network (CNN)

Where the layers highlighted in orange are processing layers with trainable parameters which need to be trained and stored while the green do not possess trainable parameters. In general, we can split the pipeline in a filtering phase composed of mostly convolutional layers, where we learn which feature to extract from the raw data, and a classification phase composed mainly of dense (or fully connected) layers.

6.1 Convolutional layers

In generic convolutional layer we have 3 main components to consider, the *Input Feature Map* (a.k.a. the input), the *Filters* and the *Output Feature Map* (a.k.a. the output) With the generic dimension showed in figure we can easily compute both the memory (bytes) and computational demand (MACs)

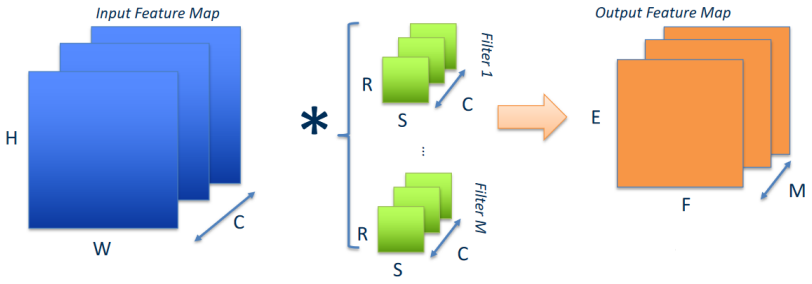

Figure 10: Convolutional layer

Naming C as the number of channels, M as the number of filter, U as the stride and being $E = (H - R + U)/U$ and $F = (W - R + U)/U$ the output dimension we get the memory requirements

as:

$$N[bytes] = M \times R \times S \times C + M \tag{8}$$

While the computational requirements (neglecting the bias) for the inference are:

$$N[MACs] = E \times F \times R \times S \times C \times M \tag{9}$$

We can immediately see how the memory requirements do not depend on the input size while the MAC have an indirect dependence due to the output size.

6.2 Dense layers

The dense layers are much simpler, given the dimensions shown in figure we can immediately say:

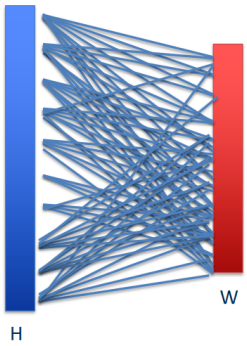

Figure 11: Dense layer

$$N[bytes] = H \times W + W \tag{10}$$

$$N[MACs] = H \times W \tag{11}$$

6.3 Feature maps

We must also consider the feature map occupation, considering a generic convolutional layer with the dimension of point 6.1 we would get

$$N_C[bytes] = M_i + M_o = H \times W \times C + E \times F \times M \tag{12}$$

Instead for a dense layer:

$$N_D[bytes] = M_i + M_o = H + W \tag{13}$$

In reality, we only need to store a maximum of 2 feature maps at a time so by considering the biggest 2 consecutive maps we can reduce this memory occupation.

6.4 Considerations

In general in a deep learning pipeline the dense layer will pose the greatest memory occupation while the Convolutional layers will amount to most of the computation. This creates an interesting trade-off since the amount of convolutional layers is usually directly proportional to the quality of the extracted features while bigger dense layers contribute to a more complex classifier.

7 Tiny architectures

Nowadays, models keep getting bigger and bigger, this is fundamentally incompatible with our application field, and we must take steps to make models run on IoT devices. In other terms we must rethink our model to fit the technological constraints especially since using a smaller architecture does not necessarily mean less accuracy.

7.1 Re-design of the CNN architecture

The first step must be to make architectural changes to existing network to make them IoT-Friendly while keeping our accuracy. We will explore three popular families of architectures for tiny devices currently employed.

7.1.1 SqueezeNet (2016)

SqueezeNet architectures were developed to: allow more efficient distributed training (since communication overhead is proportional to model size) due to the popularity of the federated learning approach; to reduce the overhead of exporting new models to the client (faster updates) and to allow for feasible FPGA and embedded deployments. As a result we want to identify smaller CNN architectures with equivalent accuracy and to redefine the design space for smaller Neural Networks.

The author outlines 3 main strategies [1]:

- Replace 3×3 filters with 1×1 filters. Given a budget of a certain number of convolution filters, we will choose to make the majority of these filters 1×1, since a 1×1 filter has 9X fewer parameters than a 3×3 filter.
- Decrease the number of input channels to 3×3 filters. Consider a convolution layer that is composed entirely of 3×3 filters. The total quantity of parameters in this layer is (number of input channels) * (number of filters) * (3*3). So, to maintain a small total number of parameters in a CNN, it is important not only to decrease the number of 3×3 filters (see Strategy 1 above), but also to decrease the number of input channels to the 3×3 filters.
- Downsample late in the network so that convolution layers have large activation maps. In a convolutional network, each convolution layer produces an output activation map with a spatial resolution that is at least 1×1 and often much larger than 1×1. The height and width of these activation maps are controlled by: (1) the size of the input data (e.g. 256×256 images) and (2) the choice of layers in which to downsample in the CNN architecture.

These 3 strategies are then condensed in the *Fire Module* defined as follows. A Fire module comprises: a squeeze convolution layer (which has only 1×1 filters), feeding into an expand layer that has a mix of 1×1 and 3×3 convolution filters. The liberal use of 1×1 filters in Fire modules is an application of Strategy 1. We expose three tunable dimensions (hyperparameters) in a Fire module: $s1x1$, $e1x1$, and $e3x3$. In a Fire module, $s1x1$ is the number of filters in the squeeze layer (all 1×1), $e1x1$ is the

number of 1×1 filters in the expand layer, and $e3x3$ is the number of 3×3 filters in the expand layer. When we use Fire modules we set $s1x1$ to be less than $(e1x1 + e3x3)$, so the squeeze layer helps to limit the number of input channels to the 3×3 filters, as per Strategy 2.

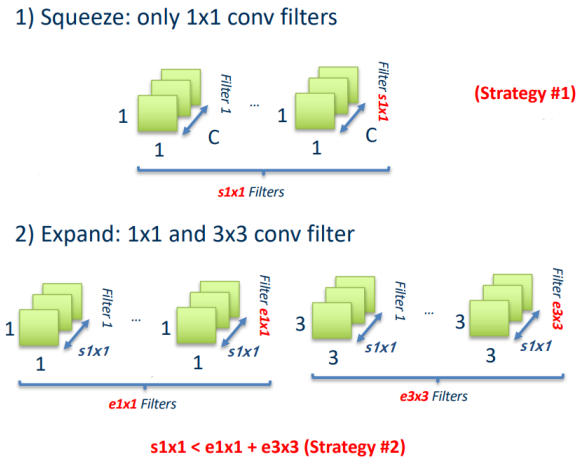

Figure 12: Fire Module

Using AlexNet as a case study we can see how this practices can achieve a 510x reduction in size without any accuracy loss, as shown in the next figure:

CNN architecture	Compression Approach	Data Type	Original → Compressed Model Size	Reduction in Model Size vs. AlexNet	Top-1 ImageNet Accuracy	Top-5 ImageNet Accuracy
AlexNet	None (baseline)	32 bit	240MB	1x	57.2%	80.3%
AlexNet	SVD (Denton et al., 2014)	32 bit	240MB → 48MB	5x	56.0%	79.4%
AlexNet	Network Pruning (Han et al., 2015b)	32 bit	240MB → 27MB	9x	57.2%	80.3%
AlexNet	Deep Compression (Han et al., 2015a)	5-8 bit	240MB → 6.9MB	35x	57.2%	80.3%
SqueezeNet (ours)	None	32 bit	4.8MB	**50x**	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	8 bit	4.8MB → 0.66MB	**363x**	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	6 bit	4.8MB → 0.47MB	**510x**	57.5%	80.3%

Figure 13: Fire Module

This particularly important in the framework of sustainability since fewer parameters and less computing time reduces the power needed to run an application. Further optimization can be applied by means of simple or complex bypass. Where the *Simple bypass* architecture adds bypass connections around Fire modules 3, 5, 7, and 9, requiring these modules to learn a residual function between input and output while the *Complex bypass* adds a bypass that includes a 1×1 convolution layer with the number of filters set equal to the number of output channels (Extra parameters to be trained). However, results only support the use of the simple bypass due to the added parameters of the complex one.

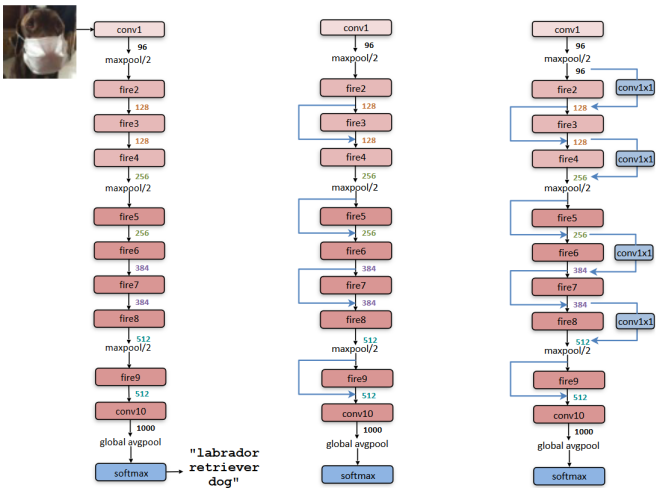

Figure 14: Macro architectural view of our SqueezeNet architecture. Left: SqueezeNet; Middle: SqueezeNet with simple bypass; Right: SqueezeNet with complex bypass

7.1.2 MobiletNet (2017)

MobileNet represent both a class of efficient models for mobile and embedded vision applications and a streamlined architecture based on depth-wise separable convolutions to build lightweight CNNs. The principles behind this architecture are represented by two main components: *Depthwise separable convolution* and a set of *Hyperparameters*.

Depthwise separable convolutions is a form of factorized convolutions which factorize a standard convolution into a depthwise convolution and a 1×1 convolution called a pointwise convolution. For MobileNets the depthwise convolution applies a single filter to each input channel. The pointwise convolution then applies a 1×1 convolution to combine the outputs the depthwise convolution. A standard convolution both filters and combines inputs into a new set of outputs in one step. The depthwise separable convolution splits this into two layers, a separate layer for filtering and a separate layer for combining. This factorization has the effect of drastically reducing computation and model size [2]. A visual representation is shown in figure.

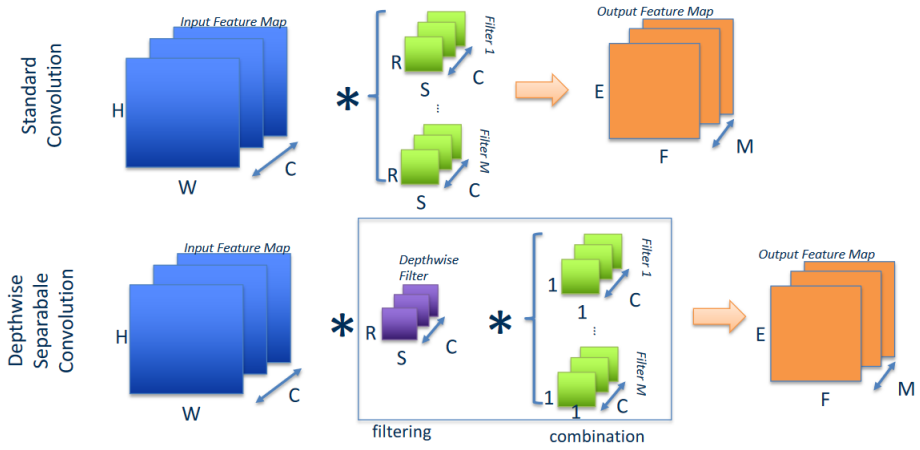

Figure 15: Visual representation of Depthwise separable convolution

Following the figure notation we can compile a table comparing the memory and MACs requirements of the standard and the depthwise convolution:

Type	MACs	Memory
Standard	$R \times S \times C \times E \times F \times M$	$R \times S \times C \times M$
Depthwise	$R \times S \times C \times E \times F + C \times M \times E \times F$	$R \times S \times C + C \times M$
Ratio	$\frac{1}{M} + \frac{1}{R \times S}$	$\frac{1}{M} + \frac{1}{R \times S}$

Table 4: Comparison between types of convolution

As we can see the Depthwise separable approach allows us to have a considerable reduction in both MAC operations and number of weights. Moreover, with the help of the hyperparameters α, or width multiplier, and ρ, or resolution multiplier, we can scale our model more efficiently acting directly on the number of filter (α) and/or on the feature map size (ρ).

The second revision of this family of network introduces the concepts of *Linear Bottlenecks* and *Inverted Residuals*.

The first concept relies on the fact that our *Manifold of Interest (MoI)*, where the interesting information resides, should be processed as part of a higher dimensional space (More channels) to avoid information loss when passing through a non-linear activation. This can be achieved efficiently by decompressing the input feature map with a pointwise convolution, then processing it, and at the end reducing the dimension using a linear bottleneck (a pointwise convolutional layer without activation). The expansion factor t has an optimal value of 6 (experimental) and indicates how much the input should be expanded.

The second concept is a simple bypass that helps the gradient propagate through the model. The nomenclature inverted comes from the fact that normally in other model residuals connections are

made between thick layers while here the layer are thin. The following figure shows the structure of this module:

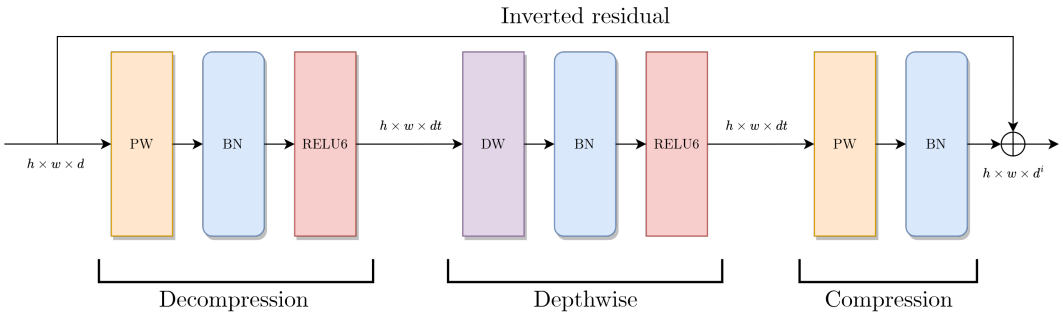

Figure 16: Linear bottleneck and inverted residual module

This allows us to reduce the number of filters for the intermediate layers, speeding up computations and decreasing the memory footprint. This second version is faster, smaller and more accurate than the first.

7.1.3 EfficentNet (2019)

The outstanding question now is how can we create scalable models to fit various technological constraints. A proposed solution is proposed in [3] taking the name of EfficentNet. The goal is to *rethink model scaling for convolutional neural networks* in order to create efficient architectures and scaling mechanism.

We can act on 3 main hyperparameters: the depth of the network d (i.e. the number of layers), the width w (i.e. the dimension of the intermediate feature maps) and the resolution r (i.e. the input resolution).

We empirically observe that different scaling dimensions are not independent. Intuitively, for higher resolution images, we should increase network depth, such that the larger receptive fields can help capture similar features that include more pixels in bigger images. Correspondingly, we should also increase network width when resolution is higher, in order to capture more fine-grained patterns with more pixels in high resolution images. These intuitions suggest that we need to coordinate and balance different scaling dimensions rather than conventional single-dimension scaling

This creates the need for a *Compound scaling mechanism* as shown in the next figure.

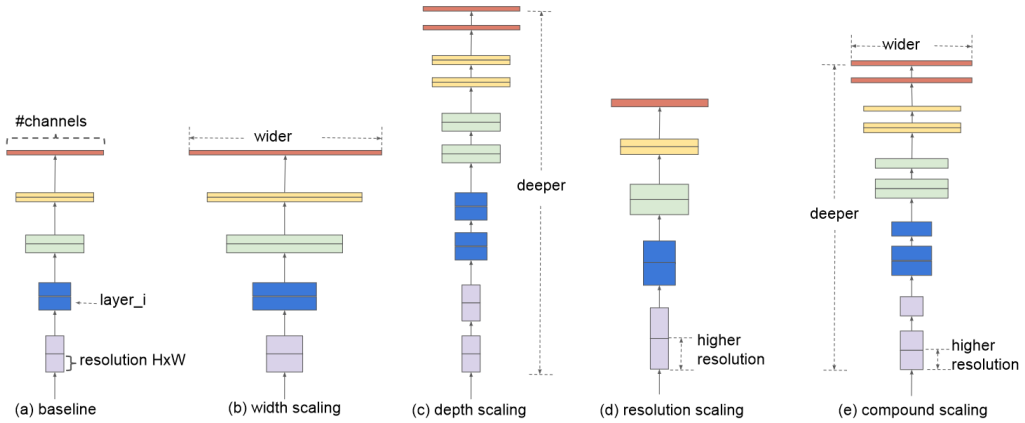

Figure 17: Scaling of CNN

Defining now α, β, γ as constants and ϕ as user-specified coefficient controlling how many more resources are available for model scaling we can define as system of equation:

$$\begin{cases} d &= \alpha^\phi \\ w &= \beta^\phi \\ r &= \gamma^\phi \end{cases} \quad \underrightarrow{with} \quad \begin{array}{l} \alpha \cdot \beta \cdot \gamma \simeq 2 \\ \alpha \geq 1, \beta \geq 1, \gamma \geq 1 \end{array} \tag{14}$$

Now by setting $\phi = 1$ setting a NN and assuming twice more resources available, a small grid search of α, β, γ is carried out, under the constraints. After we can use the resulting value of the constants and scale up the baseline network with different values of ϕ. The result are quite good as shown in the next table.

Model	Top-1 Acc.	Top-5 Acc.	#Params	Ratio-to-EfficientNet	#FLOPs	Ratio-to-EfficientNet
EfficientNet-B0	**77.1%**	**93.3%**	**5.3M**	**1x**	**0.39B**	**1x**
ResNet-50 (He et al., 2016)	76.0%	93.0%	26M	4.9x	4.1B	11x
DenseNet-169 (Huang et al., 2017)	76.2%	93.2%	14M	2.6x	3.5B	8.9x
EfficientNet-B1	**79.1%**	**94.4%**	**7.8M**	**1x**	**0.70B**	**1x**
ResNet-152 (He et al., 2016)	77.8%	93.8%	60M	7.6x	11B	16x
DenseNet-264 (Huang et al., 2017)	77.9%	93.9%	34M	4.3x	6.0B	8.6x
Inception-v3 (Szegedy et al., 2016)	78.8%	94.4%	24M	3.0x	5.7B	8.1x
Xception (Chollet, 2017)	79.0%	94.5%	23M	3.0x	8.4B	12x
EfficientNet-B2	**80.1%**	**94.9%**	**9.2M**	**1x**	**1.0B**	**1x**
Inception-v4 (Szegedy et al., 2017)	80.0%	95.0%	48M	5.2x	13B	13x
Inception-resnet-v2 (Szegedy et al., 2017)	80.1%	95.1%	56M	6.1x	13B	13x
EfficientNet-B3	**81.6%**	**95.7%**	**12M**	**1x**	**1.8B**	**1x**
ResNeXt-101 (Xie et al., 2017)	80.9%	95.6%	84M	7.0x	32B	18x
PolyNet (Zhang et al., 2017)	81.3%	95.8%	92M	7.7x	35B	19x
EfficientNet-B4	**82.9%**	**96.4%**	**19M**	**1x**	**4.2B**	**1x**
SENet (Hu et al., 2018)	82.7%	96.2%	146M	7.7x	42B	10x
NASNet-A (Zoph et al., 2018)	82.7%	96.2%	89M	4.7x	24B	5.7x
AmoebaNet-A (Real et al., 2019)	82.8%	96.1%	87M	4.6x	23B	5.5x
PNASNet (Liu et al., 2018)	82.9%	96.2%	86M	4.5x	23B	6.0x
EfficientNet-B5	**83.6%**	**96.7%**	**30M**	**1x**	**9.9B**	**1x**
AmoebaNet-C (Cubuk et al., 2019)	83.5%	96.5%	155M	5.2x	41B	4.1x
EfficientNet-B6	**84.0%**	**96.8%**	**43M**	**1x**	**19B**	**1x**
EfficientNet-B7	**84.3%**	**97.0%**	**66M**	**1x**	**37B**	**1x**
GPipe (Huang et al., 2018)	84.3%	97.0%	557M	8.4x	-	-

We omit ensemble and multi-crop models (Hu et al., 2018), or models pretrained on 3.5B Instagram images (Mahajan et al., 2018).

Figure 18: Performance of EfficentNet

7.2 Approximate computing

Starting from the framework of an efficient neural network we can introduce the topic of *Approximate Computing* to make our models fit the memory and computational requirements of tiny devices. There are two main approaches:

7.2.1 Quantization (Precision scaling)

Quantization is the process of reducing the memory occupation of the CNN by changing the precision of the weights (Number of bits for the representation). This procedure is carried out by a quantizer which can be implemented in a few different ways:

- *Linear quantizer*: uniform distance between each quantization level (number of bits: 32, 16, 8 or custom)
- *Log function*: the distance between the levels varies (e.g., log base2 quantization with logarithmic distribution of intervals). When weights are power of 2, multiplication maps into bit shifts
- *Data-driven*: quantization levels are determined or learned from the data, e.g., using k-means clustering.

The specific implementation depends on the distribution of weights as shown in figure:

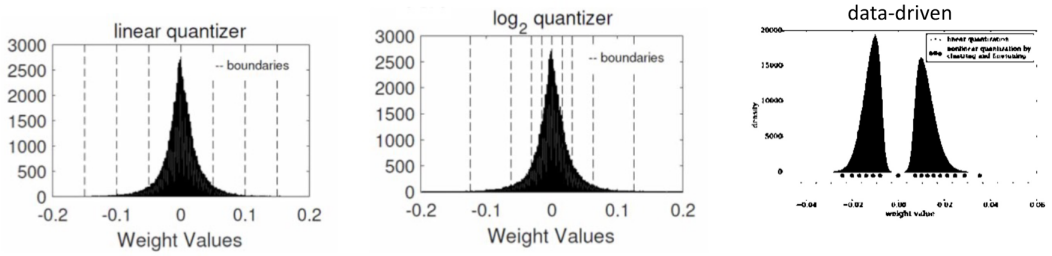

Figure 19: Different quantizer implementations

Inside the model we can quantize both the weights (reducing the memory requirements) and the activations (effect depends on the network). Additionally, we can apply Variable quantization with different quantization mechanics (w.r.t. layers) instead of a fixed quantization policy.

The performance loss attributed to quantization varies from model to model, but we can take AlexNet as a case study.

Reduce Precision Method		bitwidth		Accuracy loss vs. 32-bit float (%)
		Weights	Activations	
Dynamic Fixed Point	w/o fine-tuning [123]	8	10	0.4
	w/ finetuning [122]	8	8	0.6
Reduce Weight	BinaryConnect [129]	1	32 (float)	19.2
	Binary Weight Network (BWN) [131]	1*	32 (float)	0.8
	Ternary Weight Network (TWN) [133]	2*	32 (float)	3.7
	Trained Ternary Quantization (TTQ) [134]	2*	32 (float)	0.6
Reduce Weight and Activation	XNOR-Net [131]	1*	1*	11
	Binarized Neural Network (BNN) [130]	1	1	29.8
	DoReFa-Net [122]	1*	2*	7.63
	Quantized Neural Network (QNN) [121]	1	2*	6.5
	HWGQ-Net [132]	1*	2*	5.2
Nonuniform Quantization	LogNet [119]	5 (conv), 4 (fc)	4	3.2
	Incremental Network Quantization (INQ) [138]	5	32 (float)	-0.2
	Deep Compression [120]	8 (conv), 4 (fc)	16	0
		4 (conv), 2 (fc)	16	2.6

*Not Applied to First and/or Last Layers.

Figure 20: Quantization performance loss on AlexNet

As we can see the performance loss in very small for conservative quantization. It is important to point out that even if extreme quantization methods are listed we are limited by the specific software implementation, since most microcontrollers and devices support a minimum of 8-bit for data type.

Still, it is interesting how the ternary network $(-w, 0, w)$ for which a different scale is trained for each weight, exhibits only a 0.6% loss.

In general 8-bit quantization allows for a reduction of 3x to 30x energy with respects to an add operation and of 15.5x to 18.5x with respects to a multiply operation while also allowing for 4 operation in one system clock cycle.

In the implementation we need to specify 3 parameters: the bit width b, the scale factor s and the zero-point z (only necessary for asymmetric quantization). The scale factor and the zero-point are used to map a floating point value to the integer grid, whose size depends on the bit-width. The scale factor is commonly represented as a floating-point number and specifies the step-size of the quantizer. The zero-point is an integer that ensures that real zero is quantized without error. For asymmetric quantization the quantized value results in:

$$x_{int} = clamp(\left\lfloor \frac{x}{s} \right\rceil + z; 0; 2^b - 1) \rightarrow clamp(x, a; b) = \begin{cases} a, & x < a \\ x, & a \leq x \leq b \\ b, & x > c \end{cases} \tag{15}$$

To quantize we can use:

$$x \approx \hat{x} = s(x_{int} - z) \in [q_{\min}; q_{\max}] \rightarrow \begin{cases} q_{\min} = & -sz \\ q_{\max} = & s(2^b - 1 - z) \end{cases} \tag{16}$$

Intuitively any value that falls out of the range$[q_{\min}; q_{\max}]$ will result in clipping error. We can reduce this by increasing the scale factor s allows to expand the quantization range. However, increasing the scale factor leads to increased rounding error $\epsilon_r \in [-\frac{s}{2}; \frac{s}{2}]$

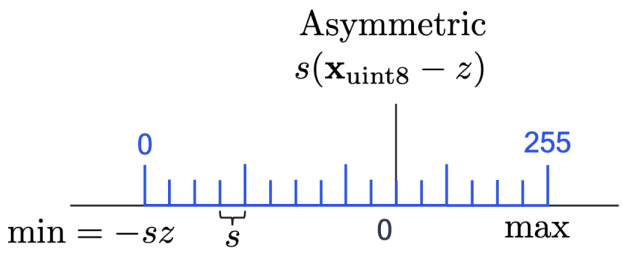

Figure 21: Asymmetric quantization

Symmetric quantization is simpler, showing both signed and unsigned version:

$$\begin{array}{lll} x_{int} = & clamp(\left\lfloor \frac{x}{s} \right\rceil; 0; 2^b - 1) & for\ unsigned\ integers \\ x_{int} = & clamp(\left\lfloor \frac{x}{s} \right\rceil; -2^{b-1}; 2^{b-1} - 1) & for\ signed\ integers \end{array} \tag{17}$$

While the dequantization step is simply $\hat{x} = sx_{int}$.

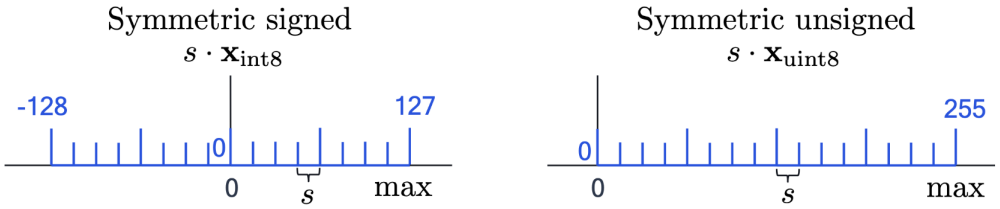

Figure 22: Symmetric quantization

Using the Symmetric quantization as an example we can write the quantized version of a MAC operation as:

$$\begin{array}{ll} \hat{A}_n & = \hat{b}_n + \sum_m \hat{W}_{m,n}\hat{x}_m \\ & = \hat{b}_n + \sum_m (s_w W^{int}_{n,m})(s_x x^{int}_m) \\ & = \hat{b}_n + s_w s_x \sum_m W^{int}_{n,m} x^{int}_m \end{array} \tag{18}$$

Where we employ two different scale factors for weights and activations. The result need to be requantized due to add operation.

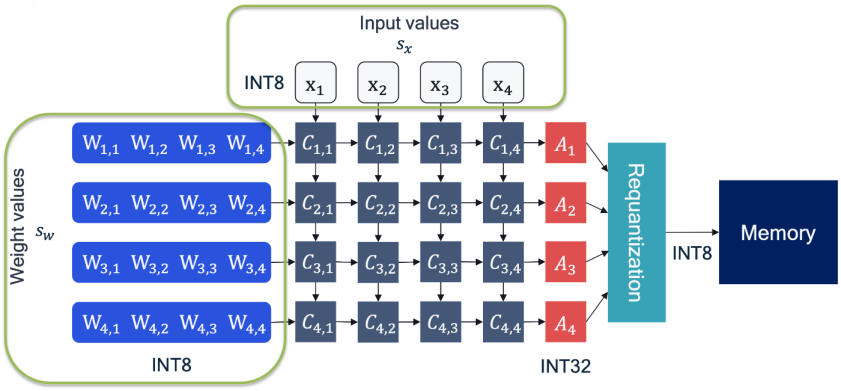

Figure 23: Example of quantized operation

Finally, we must choose between:

- Post-training: quantization (PTQ): a pre-trained FP32 network is directly converted into a fixed-point network without the need for the original training pipeline. The main challenge is the definition of the quantization range settings. Two main approaches:

$$\begin{aligned} MinMax &= [q_{\min}; q_{\max}] = [\min V; \max V] \\ MSE &= \underset{q_{\min}, q_{\max}}{argmin} \left\| V - \hat{V}(q_{\min}, q_{\max}) \right\|_F^2 \end{aligned} \tag{19}$$

  This approach is effective and fast to implement since they do not require retraining of the network with labeled data, but offers limited performance when low-bit quantization of activations (e.g., 4-bit and below) is considered and due to high rounding errors when considering wide ranges. Also, post-training techniques may not be enough to mitigate the large quantization error incurred by low-bit quantization.

- Quantization-aware training (QAT): quantization and training are jointly addressed (back-propagation implemented with simulated quantization within the training pipeline). Here higher accuracy comes with the usual costs of neural network training, i.e., longer training times, need for labeled data and hyperparameter search. Now the challenge is how to back-propagate through the simulated quantizer block. The solution is to approximate the gradient using the straight-through estimator (Bengio et al. 2013), which approximates the gradient of the rounding operator as 1 ($\frac{\partial \lceil y \rfloor}{\partial y} = 1$)

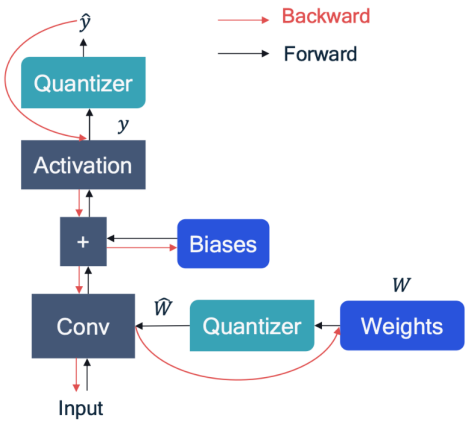

Figure 24: Quantization-aware training (QAT)

7.2.2 Task Dropping

Task dropping is the process of reducing the computational load and the memory occupation by skipping the execution of some tasks associated with the processing chain. We want to reduce the number of operations and model size. This might have a positive effect on computational demand and memory footprint. We have a wide range of techniques such as:

- Network pruning: Usually, to allow easier training, Neural Networks are over parameterized. As a result, a large amount of the weights in a network is redundant and can be removed (i.e., set to zero). However, aggressive network pruning often requires some fine-tuning of the weights to maintain the original accuracy.

  This procedure can be carried out in a more structured manner by pruning entire group of weights (entire row, entire column, filters, etc.).

- Network architecture design: In most networks we can replace a large filter with a series of small filters, either before training, by switching from 5×5 to 3×3 filter and decomposing a N×N in a combination of 1×N and N×1, or after training, by employing tensor decomposition (to decompose filters without impacting accuracy), low rank approximation and canonical polyadic decomposition.

- Transfer Learning: In a general setting we can take an existing pre-trained network, cut the last k layer and replace them with a unique processing layer. This will allow us to get an application-specific and approximated CNN without the training overhead required to retraining the whole network. In layman terms we are keeping the weights of only part of the network.

- Knowledge distillation: In this approach we employ two models, a bigger pre-made *Teacher model* and our smaller *Student model*. We can now use the combined Cross-entropy gradient to update the network. The combined gradient can be expressed as:

$$\frac{\partial C}{\partial z_i} = \frac{1}{T}(q_i - p_i) = \frac{1}{T}\left(\frac{e^{\frac{z_i}{T}}}{\sum_j e^{\frac{z_j}{T}}} - \frac{e^{\frac{v_i}{T}}}{\sum_j e^{\frac{v_j}{T}}}\right) \tag{20}$$

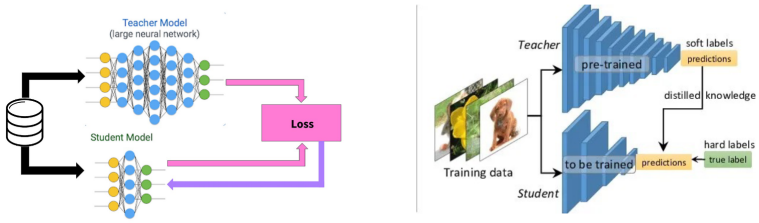

Figure 25: Knowledge distillation

7.2.3 Summary

In summary, we can define a tool chain as shown in the next figure:

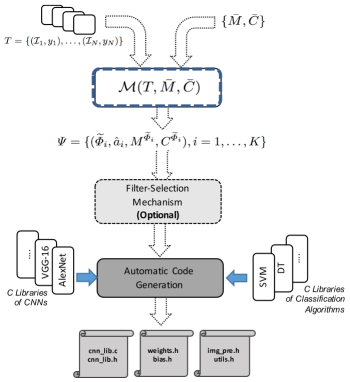

Figure 26: Tiny Model tool chain

Where M receives in input a dataset T and the constraints on memory M and computation C. Its output is the Pareto set of satisfying the constraints on M and C. The Filter-Selection Mechanism then identifies those filters providing output features that are truly beneficial to support the classification by μ and discard the others. As we can see in the next figure this can efficiently scale down a network to make it run decently even on a microcontroller:


Platform	STM32F7	Raspberry Pi 3B		
CPU	ARM M7 @ 167 MHz	ARM11 @ 1.2 GHz		
RAM	512 KB	1024 MB		
M	52 KB	102 KB		
C	100.0×10^6	100.0×10^6		
CNN Φ	AlexNet	AlexNet	AlexNet	AlexNet
k	5	5	5	5
q	0	0	0	4
Filter-selection mechanism	yes	yes	no	no
f	1	1	-	-
μ_Φ	Decision Tree	Decision Tree	SVM	SVM
$\hat{a}$	87.9	87.9	99.3	99.4
M^Φ	1.4 KB	1.4 KB	68 KB	34 KB
C^Φ	1.09×10^6	1.09×10^6	52.7×10^6	52.7×10^6
Exec. Time (ms)	2700	178	8687	8687

Figure 27: AlexNet approximation results

7.3 Embedded system code optimization

8 Early Exit Neural Networks (EENN)

Until now, we have designed Neural Networks as a stack of layers, in which a result is obtained only after processing the full stack. While this approach is straightforward it can lead to Over-parameterization, Extended latency and memory constraints and Over-learning (it might happen, for long pipelines, that at different points of the processing the decision changes for the worst, a shorter pipeline could fix this).

The rationale here is to understand that, for some tasks, we need only part of the model, extracting a meaningful output before the end of the pipeline. To correctly apply this we need to incrementally process the input through the model layers and take a decision as soon as *enough confidence* is gained.

To correctly employ this we can take the original network (In this scope referred as *Backbone network*) and insert additional small classifier (*Early Exit Classifiers or EECs*) in various point of the pipeline, creating an EENN.

This lead to the mitigation of many drawbacks of DNNs such as overfitting, vanishing gradient and overthinking as well as offering a significant reduction of the inference time. The idea also adds the capability of being distributed over multi-tier computation platforms by enhancing the adaptiveness to changing environments by achieving a desired trade-off between accuracy and efficiency on the fly. By having multiple outputs we also increase the interpretability of the network (since we can analyze each decision individually).

This all makes sense if we draw a parallel between human behavior and our models. In fact, some decisions might be better if we don't overthink them. This directly translates in what we discussed earlier.

8.1 EECs and the selection scheme

We can implement an EENN by simply considering that for each intermediate layer output $f_i(x)$ we can define small classifier (EEC) such that the intermediate decision at layer i is:

$$\overline{y}_i = C_i(f_i(x)) \tag{21}$$

In this way we now have a sequence of predictions, but how do we choose which prediction to use? In short an input sample exits from an EEC when enough confidence is achieved, in this case the sample is not forward propagated (the network makes an early exit). The decision is taken by the *selection scheme* which is a set of decision functions.

This class of function simply compares the confidence value C_i with a corresponding threshold θ_i that sets the minimum confidence to take the decision. In formulas:

$$D_i(x) = C_i(x) \geq \theta_i \ , \ i \in [1, N_l] \ , \ \theta \in [0, 1] \tag{22}$$

Note that for $\theta = 0$ the network always makes an early exit, while for $\theta = 1$ the network will never exit at that node.

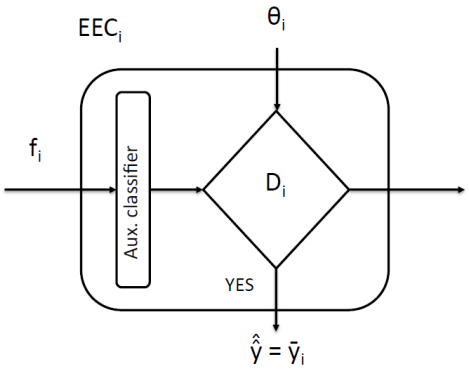

Figure 28: Early Exit Classifier

For classification problems, a popular approach for EENNs relies on ability to estimate the confidence of the neural network on its own prediction and use it to decide whether to early exit the processed input. The confidence value can be measured mainly in three ways:

- *Max Softmax output*: By directly taking the output of the softmax of the EEC we can select the class with the maximum probability and compare it to the threshold. This is a direct measure of confidence (The network does not exit if the probability is less that θ).
- *Score Margin*: The score margin (SM) is the distance between the largest and the second-largest value of the softmax output. Intuitively, the higher the "confidence" of the EEC i about its prediction, the higher the difference between the largest and the second-largest value. This is still directly comparable to the threshold.

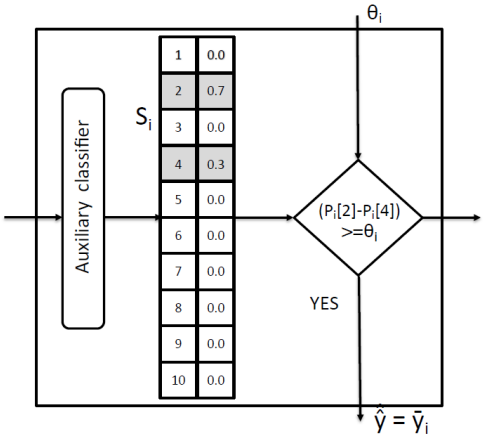

Figure 29: Score Margin Example

- *Entropy*: The entropy H estimates the level of uncertainty on the prediction:

$$H(y) = \sum_{i=0}^{N} y_i * \log(y_i) \tag{23}$$

The entropy is minimal whenever y equals a one-hot vector, and maximal when it is equal to a uniform distribution over classes.

8.2 Training of EENN

The training EENNs can be categorized into three main families:

- *Joint Training (JT)*: Consists in jointly training all the EECs by combining the loss of the classifier as:

$$L_{joint} = L(\hat{y}, y) + \sum_{i=1}^{N} w_i * L(\hat{y}_i, y) \tag{24}$$

Where L is the standard Cross Entropy loss and N is the number of EECs. The weights w_i can be set to 1 or become hyperparameters. This extends the time needed to converge as well as the data needed but trains everything at the same time.

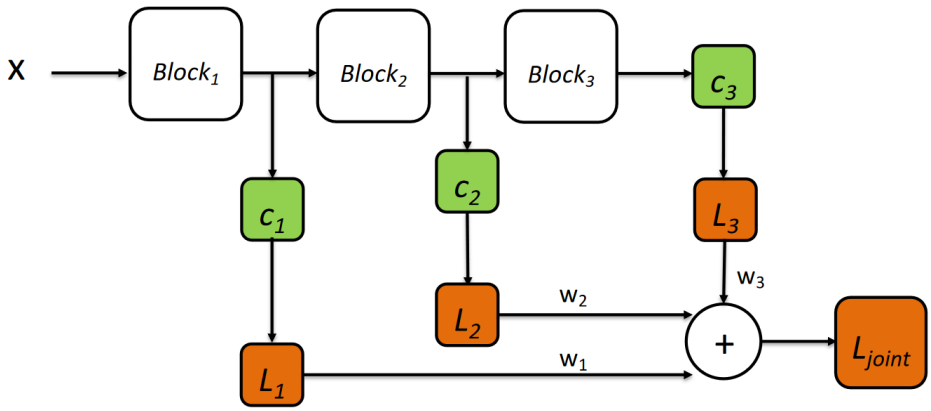

Figure 30: Joint Training

- *Layer-Wise Training (LT)*: trains iteratively one block of the backbone network and the corresponding EEC keeping frozen the former pipeline of the neural network. This does not allow each block to be the optimal feature extractor, but we only need to train one block and one EEC.
- *Knowledge Distillation (KD)*: initially trains the backbone network (teacher) and, then, the EECs (students) on top of the trained backbone network. This just means training as usual and the using the backbone as is to train the EECs. This allows for training decoupling.

Different from previous Knowledge Distillation!

8.3 Inference of EENN

We can approach inference in two ways:

- The most trivial approach is to aggregate the outputs of all the EECs to provide a joint prediction. This is a sort of *Fake EENN* since it compensates the overthinking issue by factoring in the previous decisions but does not provide any computational advantage.
- The most interesting approach is to process the input up to a given EEC and then, stop the forward propagation. This is the true reference inference scheme for EENNs.

9 Learning in presence of concept drift

Up to now we assumed the system model to be stationary and time invariant but everything and everybody changes over time. These concepts can be defined formally if consider a data generating process:

- *Stationarity*: We say that a data generating process is stationary when generated data are *i.i.d.* realizations of a unique random variable whose distribution does not change with time. This constraint is violated if the *pdf* of the data changes over time.
- *Time invariance*: We say that a process is time invariant when its outputs do not explicitly

depend on time. A time varying system for example is an AR model:

$$y(t) = a_1 y(t-1) + a_2 y(t-2) + \beta \ , \ \beta = N(0, \sigma^2) \tag{25}$$

Given that our machine learning models are built with these assumptions in mind, we must search for *Adaptation*, which in a way mimics the human behavior. Now we have two main problems: *What* we need to do to adapt and *When* we want to do it.

To answer the *What* question we propose a series of methods:

- *Instance Selection*: The idea is to identify the samples of the training set that are relevant to the current state of the process. Or in other terms to periodically change the training set to correctly adapt to the current scenario. The adaptive systems generally rely on a window over the most recent training samples to process the upcoming data which can either be:
  - *Fixed*: the length of the window is fixed a-priori by the user.
  - *Heuristic*: adapt the window length over the latest samples to maximize the accuracy

  While this method is easy to implement (low computational-complexity, reduced training set) it is also very memory hungry (We need to store the entire training set), requires a complete retraining of the model and introduces the problem of choosing the window size. Moreover, how do we choose when to change the training set?

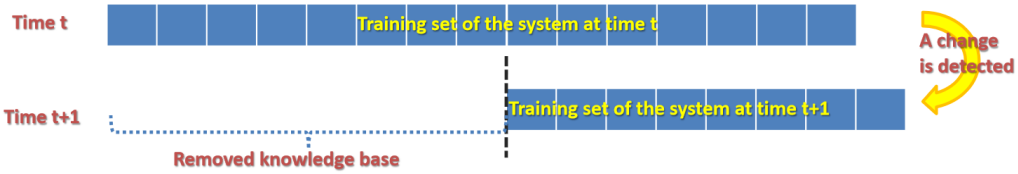

Figure 31: Instance Selection

- *Instance Weighting*: The idea is that training samples are not removed from the training set of the system but all the training samples (suitably weighted) are considered. The training samples might be weighted according to the age or the relevancy to the current state of the process in terms of accuracy of the last batch of supervised data.

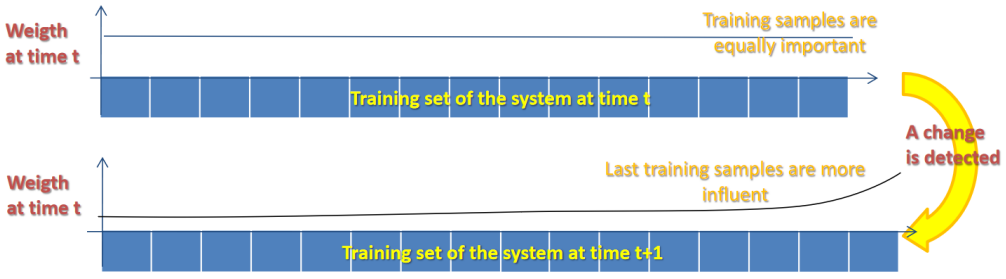

Figure 32: Instance Weighting

While this approach is still simple (low computational-complexity) and allows for all the training samples to be present at the same time is even more memory hungry and still require heuristic to define the sample weights.

- *Multiple Models*: The idea is that the outputs of an ensemble of models is combined by means of voting or weighted mechanisms to form the final output. All these systems includes techniques for dynamically including new models in the system or deleting obsolete ones (i.e., pruning techniques aiming at removing the oldest model or the one with the lowest accuracy).

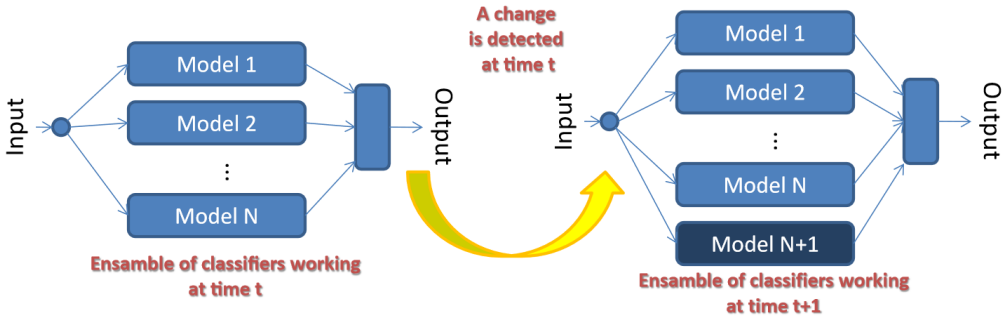

Figure 33: Multiple Models

While this approach allows us to have a model for each bunch of data it comes at the price of high computational complexity.

To answer the *When* question instead we need to distinguish between two main approaches

9.1 Passive approach

Passive solutions continuously adapt the model without the need to detect the change. This translates in ensembles of models with the adaptation phase consisting in a continuous update of the weights of the fusion/aggregation rule and creation/removal of models. This approach guarantees that the best model is always available but introduces a continuous overhead.

The general idea that the underlying data distributions may (or may not) change at any time with any rate of change, so we employ continuous adaptation of the model parameters every time new data arrive to maintain an up-to-date model at all times and avoid the potential pitfall associated with false alarms in active solutions. Some solutions may include:

Passive Learning where for each new data point or batch of data we update the weights of the model.

Ensemble learning which provides a natural fit to the problem of learning in non-stationary environments. This approach is more accurate than single classifier-based systems, can easily incorporate new data into a classification model (a new item into the ensemble) and provide a natural mechanism to forget irrelevant knowledge (removing an item from the ensemble). Some implementations are:

- *Streaming ensemble algorithm (SEA)*: where new classifiers are added as new batches of data arrive, until we reach a user-defined maximum number of classifiers, at which point we start removing some of them based on the evaluation of classifier's predictions, the age of the classifier or by simply removing the least contributing member.

- *Dynamic weighted majority (DWM)*
- *Online bagging/boosting*

9.2 Active approach

Active solutions rely on triggering mechanisms (oracle) to identify changes in the process and react by updating the model. This trigger can either monitor the pdf of the inputs (does not require labels, may ignore changes in performance), or the classification error (w.r.t. the nominal values) which is more challenging due to the need of labeled information.

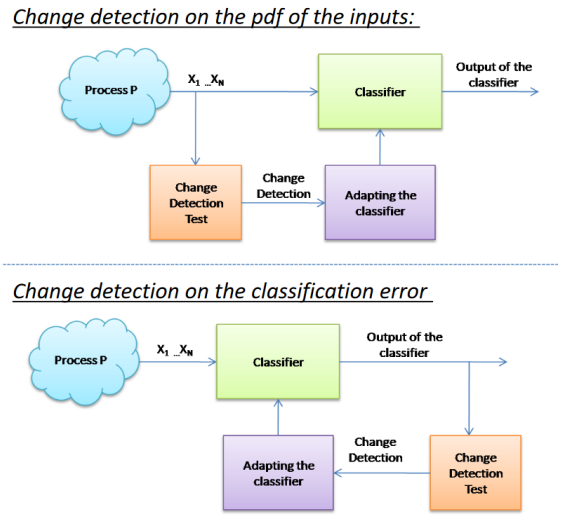

Figure 34: Monitoring schemes

For both monitoring schemes we still have to detect if a change has happened and when. In any case we can't directly analyze the data, so we must extract some features that are *iid*.

To make the process easier to understand we can define three main steps:

- *Concept drift detection*: Where ad-hoc triggers designed to detect changes by inspecting sequences of data or derived features are employed. A non-exausting list of method is provided below:
  - Data-based methods
    - Limit checking: where we test if a given (measured) variable exceeds (indicating a change) or not a known absolute limit. This method is easy to implement however the choice of the threshold is a critical issue which can lead to high/low sensitivity to change.
    - Binary threshold
  - Statistical-based methods
    - Statistical Hypothesis tests

* Change-Point Methods
* Change detection tests
  - CUMSUM tests
  - Shiryaev-Robert test
  - CI-CUSUM test, NPCUSUM test
  - ICI-based change detection test
  - Semiparametric log-likelihood criterion (SPLL)

These methods not only provide detection of the change, but also an estimation of the time instant the process becomes non-stationary. In this way by knowing the time at which the change is detected $\hat{T}$ and the estimate of the time of change T_{ref} we can estimate the real time of change T^* as shown in the figure below.

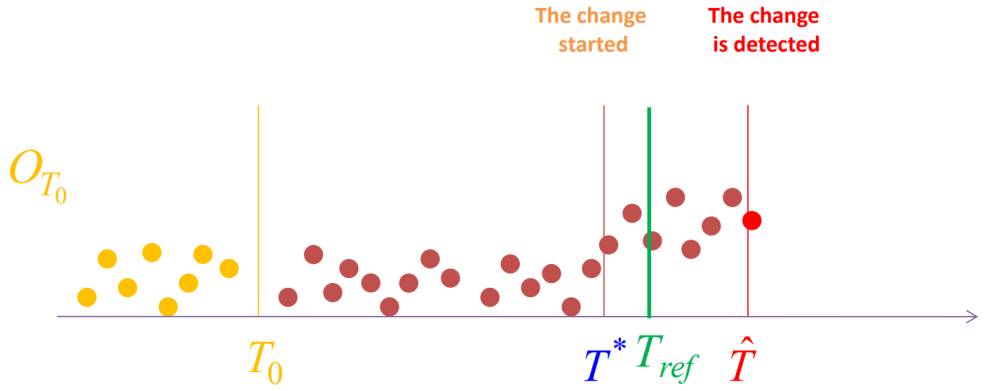

Figure 35: Change detection time estimate

- *Identification of the new state*: Additionally we want to know, based on some features, if the current state (or concept) is recurring (has already happened). This should help with ciclo-stationary processes which can have different properties based on the interval of observation.

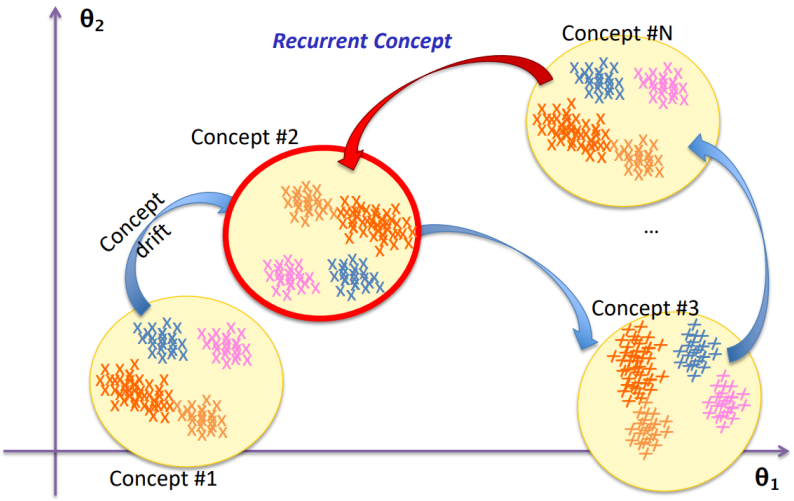

Figure 36: Recurring concepts' scenario

- *Retrain the application*: Covered in the previous section.

In general, we can make some relevant remarks:

- Being acquainted with learning techniques is a plus in everybody's background.
- Most of the time we can assume that the process generating the data is time invariant. When it is not, we need to pay attention.
- Learning in a changing environment must be considered and represents a key property intelligent systems should possess.

10 Deep Learning for IoT

In this section we want to explore how to start from the single IoT unit and get an ecosystem perspective. In general Deep-Learning for IoT concerns three macro topics such as:

10.1 On-Device Learning

While in the previous section we already discussed learning in the presence of concept drift, now we need to port everything on the device. In the IoT context the drift can be caused by:

- Changes in the environment
- Changes in the user's behavior/interest
- Faults/malfunctioning affecting the hardware or sensors/actuators
- Aging effects or thermal drifts affecting the sensors

In general perturbed, incorrect and missing data can hence heavily affect the subsequent processing phase to possibly induce wrong decisions or on-the-field reactions.

To address this issue both active and passive solutions can be employed as long as instance selection is the updating method since instance weighting and multiple models are not feasible on device.

A current challenge in the field concerns back propagation, since all the activations and errors must be stored to update the network. A proposed solution is to walk away from conventional CNN and move to trainless clustering algorithms like kNN (k Nearest Neighbors). In this solution a general feature extractor extracts features from the input and after the N-Dimensional feature space has been reduced we can use some previously stored features to run the clustering algorithms. An objection may be raised since this in not much different from decoupling the last part of a CNN, however it's difficult to identify at which layer we can extract the most significant features.

While clustering algorithms can solve some of our issues they impose significant memory demand and computational complexity.

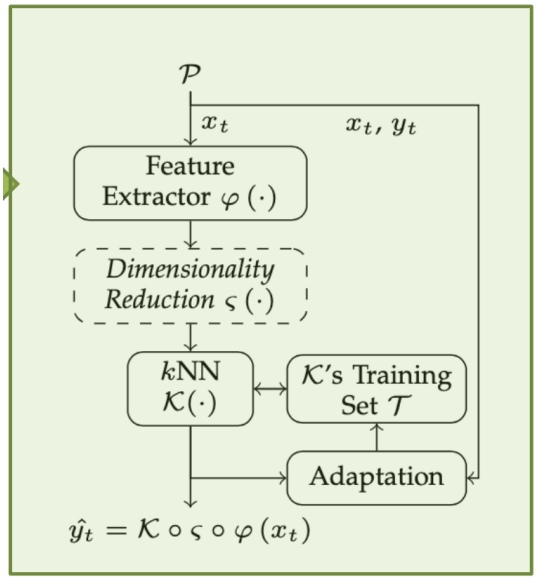

Figure 37: kNN Hybrid Architecture

Moreover, the knowledge base of the kNN is a feature space (Which is always smaller than the sample space), and can be further condensed by identifying the smallest subset of training data that can correctly classify all the training samples.

The adaptation mechanism can be:

- **Passive**: The adaptation is carried out at each new supervised Sample.
- **Active**: The adaptation is triggered by a CUSUM-based change-detection mechanism.
- **Hybrid**: Where the hybrid update continuously adapts the knowledge base (T) over time thanks to the passive adaptation while the active adaptation present in the hybrid update can quickly discard obsolete knowledge when a change is detected and set a bound on the memory footprint of the knowledge base.

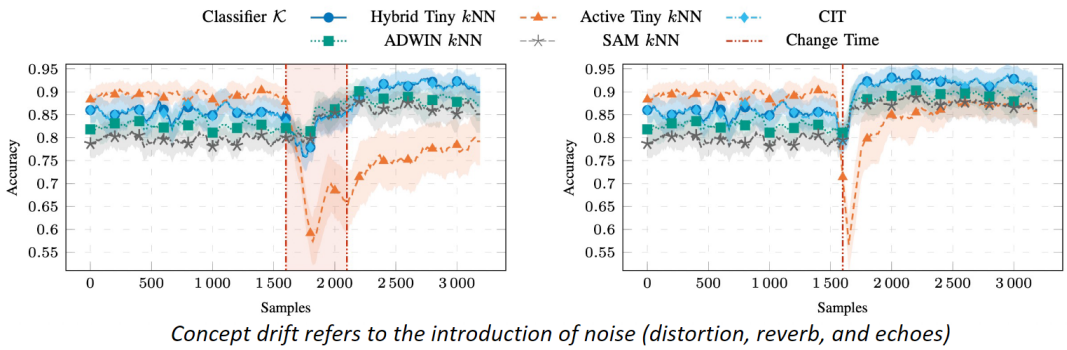

Figure 38: kNN Hybrid Architecture Performance

This topic is still actively discussed in the research space in particular regarding *Semi-Supervised TinyML* to remove the need for labels in the training and classification process and *Automatic design, development and deployment for On-line Tiny Machine Learning*

10.2 Distributed CNNs

While until now we have explored how to reduce the size of CNNs to fit our tiny devices another approach may be useful when IoT is involved. In particular, we want to exploit the ability of an IoT node to communication with its neighbors by assigning each devide a single layer of Network. In this way each node in the system (which can include different platforms) can execute the layer and transmit the results to next node. Now the problem is how to identify the optimal allocation of CNN layers on a set of heterogeneous IoT units since on a single network we can run multiple models independently (figure on the left) or share some resources if the models have common layers (figure on the right).

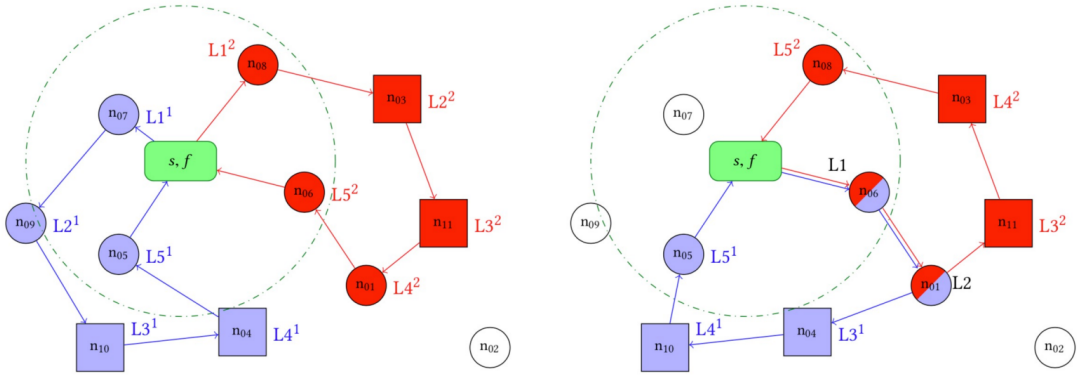

Figure 39: Examples of Distributed CNNs (Different colors indicate different models)

With this framework the possibilities are endless, however we should keep in mind the transmission overhead both in terms of power and latency.

10.3 Federated Learning

Federated learning is a decentralized machine learning approach that enables collaborative model training while preserving data privacy. It addresses the challenges associated with centralizing data from multiple sources by allowing individual participants to train a shared model using their local data. The process involves iterative rounds of communication between a central server and distributed devices, such as smartphones or edge devices, where the data resides.

In federated learning, during each communication round, the central server sends the current model parameters to the participating devices. The devices then perform local model updates using their respective data while keeping the data itself locally stored and secure. The locally updated models are then aggregated by the central server, either by averaging or other aggregation methods, to generate a new global model. This process ensures that the server gains insights from the combined knowledge of the distributed devices without accessing their raw data.

The key advantages of federated learning lie in its privacy-preserving nature and its ability to accommodate data distributed across various devices. By keeping data local, federated learning reduces privacy risks associated with data transmission and storage by employing Homomorphic Encryption, Secure Multi-Party Computation and Differential Privacy (keep in mind that is still possible to extract the training data from the model). Additionally, it overcomes challenges related to data silos and enables model training on edge devices with limited connectivity. This decentralized approach allows for broader participation, as data owners can contribute to the model training process while retaining control over their sensitive information.

This approach is particularly useful in scenarios where we deal with:

- Pervasively Distributed devices (smartphones, IoT devices).
- Privacy Issues, Sensitive Data (personal, medical, banking data).
- User-needs require cooperation (data and computational power).
- System Heterogeneity (devices can be different in HW characteristics).
- Statistical Heterogeneity (possible non-independent and identically distributed (non-IID) data across the set of devices).
- Non-Stationary Environment (seasonality effects, changes in users' behavior).

Various architectures have been formulated based on the fact that devices may have different data and feature spaces:

- Horizontal: Local datasets share the same feature space, but they are different in samples.
- Vertical: The sample space of the local data sets is the same (or similar) but the feature space is different. The clients are trained with the same data, using different features.
- Federated Transfer Learning: Local datasets differ both in samples and features.

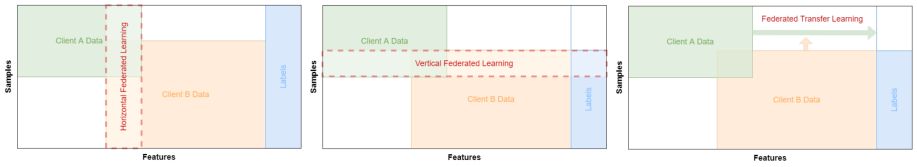

Figure 40: Federated learning architectures

For completion, we list the most popular aggregation mechanism for federate learning named *Federated Average (FedAVG)* which is an iterative process consisting in:

- 1. Server sends initialized model's parameters to the clients
- 2. Clients perform their local updates
- 3. Clients send back their updated parameters
- 4. Server aggregates the parameters (simple average)
- 5. Then, the parameters are updated and sent to the clients
- 6. Go back to point 2

References

[1] Forrest N. Iandola and Song Han and Matthew W. Moskewicz and Khalid Ashraf and William J. Dally and Kurt Keutzer, *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size*, 2016.

[2] Andrew G. Howard and Menglong Zhu and Bo Chen and Dmitry Kalenichenko and Weijun Wang and Tobias Weyand and Marco Andreetto and Hartwig Adam *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*, 2017.

[3] Mingxing Tan and Quoc V. Le *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*, 2019.