2

Edge AI Lifecycle Management

Dinu Purice[1], Francesco Barchi[2], Thorsten Röder[1], and Claus Lenz[1]

[1]Cognition Factory GmbH, Germany
[2]Universita di Bologna, Italy

Abstract

This chapter aims to define and interpret phases of the AI Lifecycle for Edge AI applications. We highlight common pitfalls that can arise when developing and maintaining AI models at the edge and outline best practices that are recognized in academia and industry, with the goal of developing a well-established taxonomy and pipeline for the lifecycle of Edge AI. We lay out that edge-based AI is seen as a natural extension of the cloud-based AI paradigm, solving problems related to real-time responsiveness, privacy, and independent operation closer to the source of the data. The challenges of edge-use cases are summarized, including limited network access, limited computational resources, and the need for customised deployment and maintenance procedures.

Keywords: machine learning (ML), software development lifecycle (SDLC), system-on-a-chip (SoC), deep learning (DL), dataset curation, edge AI, cloud-centric AI, model compression, deployment, monitoring, continuous learning, edge hardware.

2.1 Introduction and Background

In an ever more digital world, AI-based solutions have proven to be a driving force that is reshaping industries at an unprecedented pace. As artificial neural network architectures are growing, cloud-centred AI is a feasible approach to deal with increasing computational requirements as no dedicated hardware is required and cloud-based systems can scale with application demands. This also unlocked the potential of AI-based solutions in industrial applications, followed by a continuous adoption of such technologies. Certain applications such as autonomous driving, financial trading, and healthcare monitoring. impose strict latency and availability requirements, which, coupled with concerns of data privacy and bandwidth availability, result in a set of requirements that cloud computing is unable to satisfy. Incorporating local data processing can be the key to achieving fast response and real-time latency, decoupled from the inherent delays arising from device-to-cloud communication. It enables decentralized solutions capable of inferring off-line, increasing service availability while lowering bandwidth and power requirements. In addition, it improves data privacy. By delegating computation closer to the edge, relevant features can be extracted, and private ones obfuscated before any data gets transferred over a network. This also reduces the size of data transfers, further easing the requirements on the network infrastructure of the overall solution.

EdgeAI refers to the practice of doing AI computations near the users at the networks edge instead of centralised locations [1], and in this context, becomes a natural extension of the cloud-centric paradigm, enabling the transfer of computations closer to the data acquisition source [2]. The EdgeAI computing market was estimated at $9 *bn* in 2020, with projections to reach $59.6 *bn* by 2030 [3]. Following this definition, the term edge device can describe both computation nodes between the edge and the Cloud (referred to as *fog computing*)[4], and lightweight processing units coupled with the sensors acquiring the data. The second type, further referred to as *low-powered devices*, includes field-programable gate arrays (FPGAs), tensor processing units (TPUs), media processing engines (MPE), as well as other processing units.

The main areas of applications of EdgeAI are among security, mobile networks, healthcare, voice and image analysis [5]. The list of tools and devices is constantly expanding as use-cases including predictive maintenance in industrial environments: sensors for predicting asset deprecation and maintenance timeline of production chains. Developments and innovations in the field of Edge AI happen both for software and hardware hand in hand, driven by the need for specialised frameworks for low-powered devices that cannot make use of containers and virtualisation typical of Cloud-based solutions. This enables a large variety of available solutions, of diverse complexity and computational power. On the other hand, the migration of existing *Machine*

Learning (ML) algorithms to the edge faces significant challenges due to the limited hardware capabilities associated with low-powered devices.

Various techniques of compression, enabling coping with the reduced hardware capabilities are an active subject of research. Most notable among compression techniques are *pruning-based* [6], which decreases the size and complexity of trained model by eliminating non-contributing components (weight, neuron, channel, filter) with minimal impact on accuracy and *quantization-based* [7], which reduce inference complexity by switching from the standard float-32 representation to more bit-conservative ones.

Within this chapter we define the stages of the Edge AI Lifecycle by augmenting the well-established *Software Development Life Cycle (SDLC)* with ML and edge-specific processes and stages.

For convenience, all phases are grouped into three stages: (I) Pre-Development, (II) Development, and (III) Production. It should be noted that, like the SDLC case, phases in the Edge AI Lifecycle can overlap and cycle back. An overview of the flow of the Edge AI Lifecycle is presented in Figure 2.1 with examples of frameworks used at each stage.

Starting with the first phase in the Lifecycle of any software solution is the *requirement formulation* phase, which includes functional and non-functional requirements. Based on these, we introduce the *ML methodology planning* phase, which includes data planning (describing the type of data and availability of labels to be used for training, validation, testing) as well as the choice of software frameworks and ML methodologies. A difference between the Cloud-centric approach and the edge approach, is the addition of a third component in the ML planning phase, namely the selection of inference hardware. Typically, in the case of Cloud-based solutions, the developed solution enters the deployment phase completely virtualized, and able to be deployed on any of the typical Cloud hardware, customisable within a few clicks on established platforms such as AWS. This is not the case for edge solutions, when the choice of hardware imposes restrictions on the software frameworks to be used, and on data formats. In the Edge use-case this step contains three intertwined components which impose limitations on each other. Following this phase, along with the *dataset assembly* phase, is the *Development* stage, consisting of the training, validation, evaluation, and optimisation phases. The conceptual difference between validation and evaluation is that validation chooses the best performing model of the many trained, while evaluation is used to obtain a representative estimate of its performance on unseen data. Following that, the optimisation stage then simplifies the obtained model while ensuring the accuracy does not drop below the specified requirements,

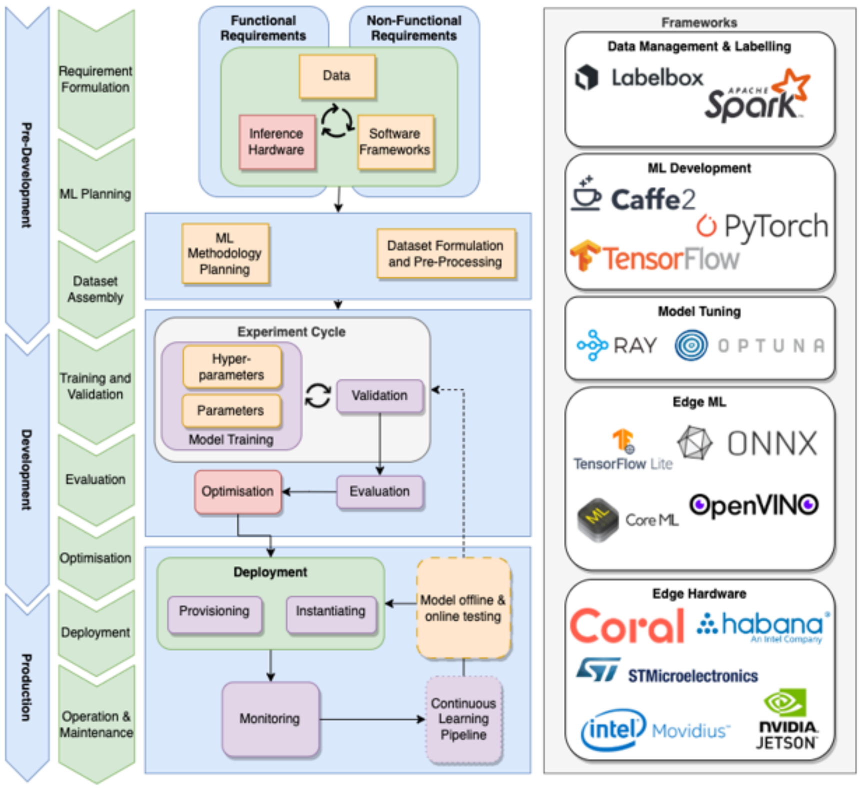

Figure 2.1 AI Lifecycle Stages Overview.

and depending on the techniques used can sometimes partially overlap with the training phase. Finally, the *Production* stage encompasses the deployment, operation, and maintenance phases. The following subchapters address each of the stages defined above and elaborate on the good practices and common pitfalls recognized within academic and industrial environments, with the goal of pushing towards an openly standardized approach to Edge AI development and deployment.

2.2 Pre-development

We define the Pre-Development stage as encompassing the **definition** and **planning** of the ML solution and associated hardware for inference, along with the assembly of the corresponding dataset.

The *definition phase* includes problem formulation, which represents the process of translating the real-world problem into a format that can be solved by a machine. For the *planning phase*, we introduced three intertwined categories in the previous sub-chapter: hardware, software, and data. To better understand the interaction between the three categories, we depart from the relationship between data types and machine learning algorithms capable of processing each type. Based on the problem statement, the type of data and the scope of the ML algorithms, several learning paradigms can be distinguished, as outlined in Table 2.1.

Each type of learning is equipped to handle different types of tasks, with their own requirements in terms of data and annotations. Unsupervised learning for example, being used mostly for extracting insights from large datasets with no labels, is rarely deployed to the edge. Typical edge use-cases refer instead to supervised (or semi-supervised, depending on the availability of labels for the data) or reinforcement learning. Every type of learning can in turn be further decomposed into different types of tasks. For example, a classification task can be formulated as binary classification, multiclass classification, or multi-label classification. A segmentation problem, very common in computer vision tasks, can be treated either as semantic segmentation (pixel-wise segmentation into foreground and background), or instance segmentation (different objects of the same class receiving distinct labels of the same class). Different formulations entail different labelling effort requirements. For example, although instance segmentation outputs

Table 2.1 Types of Learning and corresponding tasks

Type of Learning	Explanation	Application tasks
Supervised	Learning a function that maps an input to an output based on sample input-output pairs (labelled data)	Classification, Regression, Semantic Segmentation, Instance Segmentation
Unsupervised	Analyses unlabelled datasets without the need for human labelling (data-driven)	Feature Extraction, Trend identification, Clustering, Principal Component Analysis
Semi-supervised	Represents a combination of the above two types, typically used in dealing with a partially labelled dataset	Can be used to tackle both supervised and unsupervised type tasks
Reinforcement	Attempts to evaluate the optimal behaviour in a particular context or environment, based on reward or penalty.	Robotics, Autonomous Driving, Natural Language Processing

a detailed mask covering all pixels belonging to an instance of the object of the given class, in some applications a separation between foreground and background would suffice, significantly cutting the time required for labelling. It is recommended to choose the simplest problem formulation type which satisfies the task requirements, with the goal of minimising the resource requirement to prepare the dataset.

Dataset Formulation

The goal of the dataset preparation phase to create a set of data that is representative of the intended use-case, with sufficient examples to provide the developed neural network model with enough space during training for generalisation and identification of relevant features. It represents a critical stage which might make the production of a high accuracy model an impossible task, or more difficult than it needs to be. Hence, the right *domain knowledge* is required. Domain knowledge refers to the general background knowledge of the field or environment from which the data originates. It is particularly important for identifying outliers and non-representative datapoints, detecting biases, and proposing attribute sampling methods, to reduce the non-informative data in the set. Equally important, domain knowledge is required for the formulation of data labelling guidelines, which help to ensure that the dataset is consistently annotated even if the annotation process is done by multiple experts as annotation variability must be kept to a minimum. Furthermore, to better combat the possibility of human error, data labelling by multiple experts in parallel can be an effective, albeit costly, solution. One common pitfall arising from insufficient data analysis and lack of domain knowledge during the pre-development stage is *concept drift* [8]. It refers to unforeseen changes in the relation between input and output data that are left unaccounted for. An example of concept drift is the shift in relation that might occur due to seasonal conditions e.g. summer to winter. Based on the nature of the change of the statistical properties of the predicted variable, the drift can be *sudden*, *gradual*, *incremental*, or *periodic*. The dangers of concept drift are amplified by the fact that its negative impacts on the accuracy are not detectable during training, and only become apparent during production, manifesting as degraded performance of the deployed solution. More on the detection and combating of concept drift is presented in the Production sub-chapter.

Data augmentation represents the process of “artificially” increasing a dataset by modifying copies of existing datapoints (*augmentation*) or synthetically generating new ones using the existing dataset (*synthetic*). Although typically used to expand datasets which have high costs of labelling, data augmentation techniques are also useful as an additional regularization factor, and to combat overfitting during training [9]. Another non-trivial use of data augmentation is to create datasets out of private data, when augmentation is used to obfuscate private features. Data augmentation can be counter-productive in cases with *data bias*, as the inherent bias in the data persists (and can be amplified) in the augmented dataset. Data bias describes the effect of over-representing certain elements in the dataset. It leads to models trained on it ending up "lazy", i.e. biased to predict the majority class.

There are multiple techniques of addressing the bias inherent in the data, at various stages of the AI Lifecycle. During pre-development, bias-compensating strategies include re-weighting and re-sampling the data, such that the dominating class becomes under-sampled. To further improve the generalisation capabilities and convergence of the developed model, it is recommended to make use of statistical rescaling techniques, such as *normalisation* and *standardisation*. Normalisation rescales the data to a [0,1] interval, and should be used when the distribution of the expected real-world data is unknown, while standardisation rescales the data such that the mean becomes zero and the standard deviation becomes one. It should be used when it can be assumed that the expected data follows a Gaussian distribution. Such techniques are helpful with improving the convergence speed during training, and with the regularisation of model weights. Before proceeding to Development, the dataset is split into training, validation, and test subsets. Most common split ratios include 60-80% for training, 10-20% for validation, and 10-20% for testing. While the train and validation subsets are actively used in the Development stage, the purpose of the test subset is to give an estimate of the performance of the resulting model on unseen data and should therefore only used for computing a final quality metric once the best performing model on the validation set is selected.

2.3 Development

In the domain of Edge AI, the development of lightweight neural network architectures has gained substantial importance. This is primarily driven by the increasing demand for precise and resource-efficient *Deep Neural Networks* (DNNs), especially in scenarios where these networks need to operate on resource-constrained edge computing devices. The development stage represents the most computationally intensive phase during which the network model is created, trained, evaluated, and optimised. At this stage in particular, comprehensive documentation is needed to record every step of model development, including weight initialisation and random number generator seed, to ensure the *reproducibility* and *transparency* of the process. Development is usually conducted in the Cloud or on-premise servers, where sufficient computational resources are available. This sub-chapter will provide a brief overview of state-of-the-art methodologies used for architecture design and training, as well as techniques for model compression and optimisation. Additionally, an overview of state-of-the-art hardware used for edge applications will be presented.

Model architecture development and training phase

Training phase represents the iterative process of exposing the neural network model to the dataset, enabling it to learn and adjusting its weights and biases, also known as model *parameters*, such that the accuracy of the model, measured on the train set (also referred to as fitting accuracy) increases. The process starts with model initialisation, during which the model *parameters* are either randomly initialised or pre-set in case of a pre-trained network, as well as with the selection of a *hyperparameter* set. The term hyperparameters refers to a broad set of choices made prior to the network training phase, and include design decisions of the network architecture (number of layers, neurons, filters, etc), learning rate, activation functions, optimisation algorithm, etc. The difference between model parameters and hyperparameters is that the first refers to the weights of the model trained through backpropagation applied on the model's loss function, while hyperparameters refer to top-level parameters controlling the learning process. Picking the right hyperparameters is not a straightforward process, and a sub-optimal choice would negatively influence the convergence of model training, as well as the resulting overall accuracy. The activity to identify suitable hyperparameters for DNN models within reasonable timeframes for novel applications has necessitated the adoption of automated pipelines. Trivial techniques for hyperparameter optimisation include *manual search*, *grid search*, and *random search*. These involve the launch of multiple experiments (i.e. independent training processes) with manually, grid-based, or randomly selected hyperparameters out of the set of possible values, which are then tried either sequentially or in parallel. The efficacy of each is then evaluated based on the performance of the model on the validation dataset, during or after training. Such methods do not guarantee that the optimal solution is found and are expensive in terms of computational resources and time. Due to the sheer complexity of manually exploring an extensive array of hyperparameter combinations, there has been a growing need for derivative network architecture search technologies. To this extent, more informed searching methods have been developed, such as *Evolutionary* and *Population-based Optimisation* [10]. These methods are adaptive, meaning they stop experiments in which the choice of hyper-parameters has proven to be sub-optimal, as measured by a user-defined fitness function. The terminated experiments are then replaced by new instances with hyperparameter sets derived from the more promising experiments. Such approaches are very efficient at minimising the training time and the hardware resources consumed compared to the previously mentioned classical search methods, and in addition provide a more exhaustive search over the hyperparameter space. Frameworks like *Ray Tune*, *Optuna*, and *Hyperopt* provide implementations for hyperparameter optimisation, and are compatible with most common ML frameworks such as *PyTorch*, *TensorFlow*, and *Keras*.

Another approach to hyperparameter optimisation is given by *Neural Architecture Search* (NAS) algorithms, which exhibit the capacity to optimize a diverse range of functions, encompassing both precision and complexity considerations, within a discrete search space. These algorithms have a considerable drawback due to the challenging evaluation step. Indeed, evaluating a sampled DNN necessitates a computationally intensive full training process.

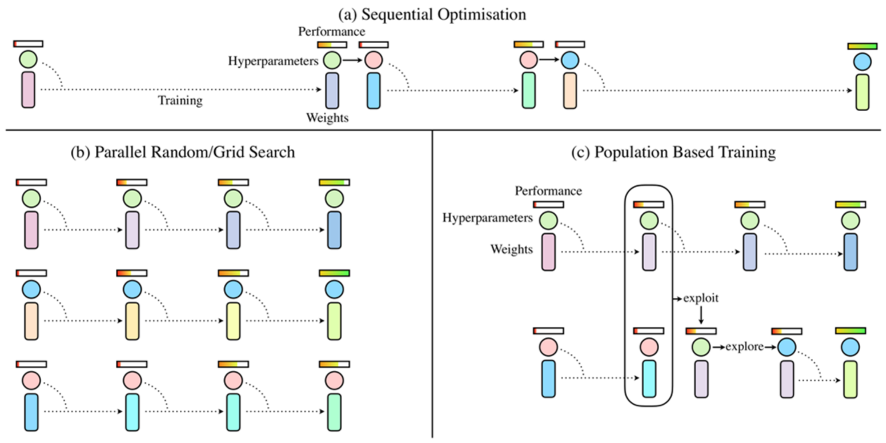

Figure 2.2 Overview of hyper-parameter training methodologies [10] illustrating (a) sequential optimisation; (b) parallel optimisation; (c) adaptive optimisation.

To alleviate this computational load, different techniques have been developed, such as the usage of reduced datasets, look-up tables and approximation of models to estimate cost-related metrics (memory occupancy, latency, energy consumption). *Differential Neural Architecture Search* (DNAS) represents a pivotal advancement in the realm of NAS, markedly reducing the time required for optimization. This reduction is achieved by transitioning from a discrete search space to a continuous one, rendering the problem addressable using gradient-descent optimization techniques. The central idea of DNAS revolves around the definition of a set of architectural parameters able to encode the selection of a DNN architecture from the search space. DNAS jointly optimizes these architectural parameters alongside the weights of the neural networks. This amalgamation of architectural parameter optimization and weight training within a continuous search space contributes to the accelerated optimization of DNN architectures, making DNAS a promising approach to exploring efficient and effective neural network design.

"[11] introduces DARTS Differentiable Architecture Search", addressing the challenges associated with scalability in architecture search. DARTS introduces the DNAS concept, framing architecture search as a differentiable problem. Through the continuous relaxation of architectural representations, DARTS enables accelerated search processes employing gradient descent techniques, significantly reducing search time. Extensive experiments have been conducted on diverse datasets, including CIFAR-10 and ImageNet, showing DARTS exceptional ability to uncover high-performance convolutional and recurrent architectures tailored specifically for image classification and language modelling tasks. This goal is especially relevant in a domain where optimized network architectures, capable of accommodating the constraints of edge devices, hold considerable importance, thus contributing to the advancement of EdgeAI model development. In [12], the researchers acknowledge the escalating demand for DNN models that strike a balance between precision and operational efficiency, a requirement in the context of edge computing. PLiNIO, is an open-source library that consolidates a comprehensive set of cutting-edge DNN design automation techniques into a user-friendly interface. These techniques, rooted in lightweight gradient-based optimization, simplify the intricacies of DNN development for edge applications. Through empirical assessments conducted on tasks pertinent to edge computing, the study demonstrates that PLiNIO yields many DNN solutions that surpass baseline models in respect of the delicate trade-off between accuracy and model size. It is worth noting that PLiNIO exhibits remarkable memory reductions, up to 94.34%, while maintaining accuracy levels, underscoring its pivotal role in EdgeAI model development.

In summary, derivative network architecture search technology, exemplified by pioneering frameworks such as DARTS, is pivotal in the EdgeAI model development. These innovative approaches make the optimization process more efficient, allowing us to navigate the complex landscape of hyper-parameter configurations and unveil DNN architectures that achieve optimal equilibrium between accuracy and model size. This research direction holds great promise for the future of Edge AI, where resource-efficient, high-performing neural network architectures serve as the bedrock for a wide range of applications.

Model validation phase

Model validation asseses the quality of the training process by measuring the accuracy of the model on a dedicated validation dataset (validation accuracy). It goes hand in hand with the training phase. Insights acquired from the validation accuracy assessment are then used to compare different training instances to identify the optimal hyperparameter choices, and to assess when the training process should be stopped. Typically, an *early stopping* mechanism is used for this purpose, which monitors the development of validation accuracy and stops the training once the accuracy reaches a plateau or starts degrading. Failing to stop a training session in time is one of the causes of *overfitting*, occurring when the model learns patterns unique to the training set that do not apply to real-world data. An overfitted model is typified by a high discrepancy between the fitting and validation accuracies and performs poorly on unseen data. Various techniques to combat the overfitting effect exist and can be grouped by mechanism as presented in Table 2.2.

Model evaluation phase

The evaluation phase starts once the training has been completed and the best performing instance of the model has been identified. The goal of this phase is to assess how well the trained model generalises to new, unseen data, thus emulating a real-world scenario. The accuracy of the model on the test dataset is measured, and serves as a final, unbiased indicator of the model's

Table 2.2 Techniques to combat overfitting.

Mechanism type	Technique description
Data-based	**More training data** – most straightforward, increase the diversity of the training data by adding additional datapoints
	Data augmentation – artificially increase the diversity of the training data through augmentation techniques and the addition of noise.
	K-fold cross validation – split the dataset into K subsets, with each subset used for validation set once while the others are used for training. Other similar cross validation techniques include stratified cross-validation, leave-one-out-cross-validation (LOOCV), etc.
Regularisation-based	**L1 and L2 Regularisation** – penalise complex model weights by adding their L1 or L2 norm to the loss function as an additional term
	Dropout [13] – randomly deactivate a fraction of neural network neurons during each training iteration
Feature-based	**Feature engineering** – manual selection, transformation, and creation of features from the original data
	Pruning – removal of parameters from a network based on their usefulness to the inference output, thus reducing the model's complexity
Inference-timed	**Model ensembling** [14] – combine predictions from multiple models to produce a single optimal predictive model
Training-timed	**Early stopping** – stops the training process once the validation accuracy stops improving

performance. The measured evaluation accuracy must not then be used to make any further decisions about the model's architecture, hyperparameters, or any other aspect of training. Doing so would represent a form of *data leakage* when information from outside the training and validation phases makes its way into the training pipeline and undermines the validity and estimated evaluation accuracy of the trained model. Instead, in case the measured evaluation accuracy does not satisfy the requirements set in the previous stage, the whole development process must be restarted, with new dataset splits.

Model compression phase

Compressing a DNN model is crucial for making it more suitable for deployment on resource-constrained devices. There are several techniques available to achieve DNN model compression, as outlined in Table 2.3.

Table 2.3 Compression Techniques

Compression Technique	Technique description
Weight Quantisation	Involves representing the model's weights with a lower bit precision than the standard 32-bit floating-point numbers. Common bit-widths include 8-bit or even lower, reducing memory and computational requirements.
Model Quantisation	Involves quantizing activations during inference. This can further reduce memory and computation requirements by using lower-precision representations for intermediate activations.
Pruning	Involves removing unimportant/low-magnitude weights or neurons from the model. These elements contribute minimally to the model's performance, so their removal can significantly reduce model size and inference time without a significant loss in accuracy.
Knowledge Distillation	Represents training a smaller student model to mimic the behaviour of a larger, more complex teacher model. This transfer of knowledge from the teacher to the student model results in a smaller and more efficient model that maintains most of the teacher's accuracy.
Knowledge Pruning	This approach combines knowledge distillation with pruning. The teacher model is first pruned to a smaller size, and then a student model is trained to mimic the pruned teacher. This results in a more compact model while maintaining the knowledge of the original, larger model.
Low-Rank Factorization	This technique decomposes the weight matrices of the model into lower-rank matrices. By doing this, you can reduce the number of parameters in the model, leading to a smaller model with less computational overhead.
Sparse Models	Sparse models are models with a substantial number of zero-valued weights. Techniques like sparse training or structured sparsity constraints can be applied to encourage weight sparsity, resulting in a more compact model.
Compact Architectures	Using model architectures designed for efficiency, such as *MobileNet*, *EfficientNet*, or *SqueezeNet*, can lead to smaller models that maintain competitive performance on various tasks.
Transfer Learning	Instead of training a model from scratch, one can use pre-trained models as a starting point and fine-tune them to the specific task at hand. This approach leverages the knowledge learned from a larger dataset and model, resulting in a smaller model customised for the specific task.

These techniques can be used individually or in combination to achieve the desired level of compression while minimizing the impact on model accuracy and performance. The choice of technique(s) depends on the specific requirements of the task at hand and the available computational resources. Quantization, for example, has evolved significantly in the context of Edge AI applications. Its history is marked by the pursuit of approximating floating-point weights and activations with low bit-width integers, ultimately aimed at reducing model size and enhancing operational efficiency. Particularly at the edge, where computational resources are constrained, quantization is a critical factor in making DNNs more viable [15]. In the past, quantization was often applied post-training, essentially mapping the high-precision model to a lower-precision representation. However, a significant breakthrough came with the introduction of *Quantization-Aware Training* (QAT) [16]. QAT enables DNNs to adapt to the effects of quantization during the training process, mitigating the subsequent drop in accuracy that occurs with post-training quantization. Standard fixed-precision quantization assigns a uniform integer bit-width to the entire DNN, neglecting the unique sensitivity of each layer to precision reduction. Recognizing this limitation, the field advanced with mixed-precision methods [17]. These approaches introduce variability in bit-width assignment, quantizing different subsets of the DNN at varying levels of precision. This innovation, however, introduces a challenging optimization problem, demanding the identification of precise bit-width assignments that strike an optimal balance between model accuracy and computational complexity. The challenge grows exponentially with the number of considered bit-widths, making it a computationally intensive effort. Several mixed-precision strategies have emerged to address this complexity-accuracy trade-off, representing a parallel development orthogonal to NAS. Additionally, some strategies employ reinforcement learning techniques to automate bit-width assignment. Recently, a gradient-based method inspired by the principles of DNAS was introduced, enabling bit-width assignment during training [18]. This method dynamically quantizes data at various precisions and selects an optimal precision during the training process. In essence, quantization techniques have witnessed a historical shift from post-training conversion to in-training adaptation, reflecting the growing importance of model efficiency in the context of Edge AI applications and the innovative approaches developed to optimize this critical aspect of DNNs.

Hardware for Edge AI

Edge AI relies on a variety of hardware components and platforms to enable efficient and real-time inference. Many edge devices, such as smartphones, IoT devices, and embedded systems, use SoCs that integrate various components like CPU, GPU, DSP, and often hardware accelerators like *Neural Processing Units* (NPUs) or *Field-Programmable Gate Arrays* (FPGAs). These compact and power-efficient chips are well-suited for running AI workloads at the edge. General-purpose CPUs are still widely used in edge devices for AI inference, especially for less demanding tasks. Many modern CPUs come with support for hardware-based vectorization and optimizations like SIMD (Single Instruction, Multiple Data) instructions to accelerate AI workloads. GPUs, originally designed for graphics rendering, are highly parallel processors that excel at performing matrix operations essential for deep learning. Edge devices equipped with GPUs can leverage their computational power for AI tasks. Specialized NPUs designed explicitly for accelerating deep learning workloads are increasingly integrated into SoCs for edge devices and provide hardware acceleration for AI inference, improving both speed and energy efficiency, but generally have higher power consumption than dedicated hardware. FPGAs offer hardware programmability, making them adaptable to specific AI models and tasks. They are commonly used in scenarios where low latency and real-time processing are crucial, such as autonomous vehicles and robotics. AI-specific accelerators, like Google's Tensor Processing Unit (TPU) and Intel's Movidius VPU, are custom-designed chips optimized for AI workloads. These accelerators are highly efficient for tasks like image recognition, object detection, and voice processing, making them valuable for Edge AI applications with stringent requirements. Depending on the specific needs of an Edge AI application, custom hardware solutions may be developed to meet unique demands, such as specialized hardware for robotics. The choice of hardware for Edge AI depends on factors such as the specific AI workload, power constraints, latency requirements, and cost considerations. Many Edge AI applications use a combination of these hardware components to optimize performance, power efficiency, and resource utilization for AI inference. Of particular significance are the architectural advancements that have emerged in recent years, owing to the advent of RISC-V, an open Instruction Set Architecture (ISA) that empowers hardware developers to devise pioneering and high-performance solutions. As an exemplar, the GreenWaves GAP8 processor, equipped with eight CV32E40P cores [19], delivers 22.65 Giga Operations Per Second (GOPS) with an exceptional power efficiency of 4.24 milliwatts per GOP (mW/GOP). This technological achievement was effectively harnessed for the autonomous navigation of a micro-drone through the execution of a neural network [20].

2.4 Production

With the increase in maturity of Machine Learning algorithms, the issue of efficient deployment and maintenance comes more and more into focus. This has led to the emergence of the MLOps field, which handles the tasks of deployment, monitoring, and operations of ML models. Provisioning, the starting point of deployment to the edge, represents the translation of the model to the specific architecture of the hardware. Within the Cloud paradigm, provisioning is less critical, as the models are traditionally deployed through virtual machines and containers, isolated from underlying hardware. Deployment to the edge however, particularly to low-power devices, requires the use of specialized frameworks, often developed and maintained by the manufacturers of said devices. Typically said frameworks consist of an intermediate representation component, which represents the prepared model in a lightweight, optimised state, and an inference engine which runs the model. Intel's *OpenVINO* toolkit is one such example, best suited for Intel's CPUs, GPUs, as well as GNAs. *CoreML* is compatible with Apple devices, while *Tensorflow Lite* is best used for Android and Coral TPUs. At the same time attempts are made at creating universal formats, such as *ONNX*, which supports a variety of frameworks used for developing ML models, such as *Tensorflow*, *PyTorch* and *Caffe*, and make them available on various hardware. Depending on the requirements and complexity of the application, the model can then be deployed to function in an online or offline mode. For describing the monitoring phase, we will assume an online functioning mode, with at least occasional network connectivity to a Cloud-based managing framework. The data used for training does not always accurately reflect the real-world encountered by a deployed model in the long term, reflected in degrading model performance over time, adjustments must be made based on insights acquired during the production stage. Variations in production data distributions, a symptom of this effect, can be detected with data drift detection algorithms, such as the *Kolmogorov-Smirnov* test, *Population Stability* Index, *Page-Hinkley* method, etc. The phenomenon of taking such insights into account and modifying the deployed model based on them is called *Continuous Learning* and represents a technique of proactive intervention to combat model drift.

After sufficient new data are acquired during production, a data curation stage is triggered, in which the data is prepared for a fine-tuning session. The fine-tunning session mirrors the training and validation pipeline, followed by an offline testing stage determining if the resulting fine-tuned model has

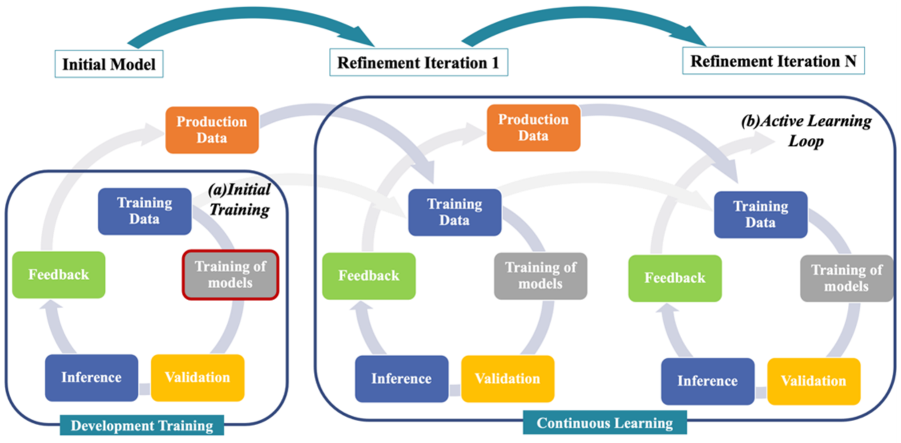

Figure 2.3 Model Training Overview illustrating (a) training during the development stage and (b) training during the production stage.

improved or downgraded its performance. Finally, the fine-tuned model is deployed in parallel to the production version, and their predictions compared in online testing, in which their comparative accuracies on unseen data are evaluated. In case the fine-tuned model is performing better, it takes the place of the previous version of the model, and the other is removed from service. Good version control is essential at this stage, to track model development and to keep the older versions as fall-back options, to be made available in case of unforeseen deviations by the active model. An overview of the ML procedures taking place within the continuous learning paradigm is presented in Figure 2.3. Here during stage (a) the optimal hyperparameters are found and the model is trained on the initial data, and in stage (b) the model makes use of the continuous learning pipeline to fine-tune its weights based on feedback from the production environment. It should be noted however that particularly in the case of Edge AI, where the production stage takes place on distributed, low-powered hardware, the infrastructure required to enable the continuous learning pipeline becomes more convoluted than in the Cloud-centric case. The issues that it needs to consider are the reduced computing power, which must be shared between the inferring component and the data acquisition component, and the limited bandwidth to be used for data transfer and model re-deployment, as well as the fact that new models must be deployed on each device. Overall, the infrastructure must support Edge-to-Cloud integration for transfers of fresh data, Model Version Control

Table 2.4 Types of Automation based on the definition by SAE International [21]

Level of Automation	**Stage Name**	**Stage explanation**
Level 0	No Automation	The human operator performs all tasks without input from the machine
Level 1	Assistance	Limited assistance is provided to the human operator in completing specific tasks
Level 2	Partial Automation	The machine takes over some of the task, but continuous human monitoring is required
Level 3	Conditional Automation	The machine can perform most tasks independently, but human intervention is required in case of complex and unexpended situations
Level 4	High Automation	The machine can perform most tasks independently, human intervention required in exceptional cases
Level 5	Full Automation	Human operator not needed at all. The machine can perform independently including in exceptional conditions

for tracking and updating models as needed, and Over-the-Air (OTA) updates for the deployed models.

Depending on the level of autonomy of the deployed AI solution, as well as the requirements of the human factor in inference monitoring, different levels of autonomy can be defined. Currently, there is a system in place for autonomy in vehicles developed by *SAE International* [21], which we will use as a starting point to generalise guidelines for the autonomy of Edge AI solutions, presented in Table 2.4.

The difficulties of reaching level three and above, as defined in the table above, particularly in the case of autonomous driving, lie in the unpredictability of the environment in which an autonomous vehicle operates. In case of controlled environments, as is usually the case for applications within factories and assembly lines, the probability of unexpected and exceptional cases diminishes considerably, easing the transition to high and full automation.

2.5 Conclusion

In this chapter we have presented Edge AI as a natural extension of the Cloud-centric AI paradigm that enables solutions for use-cases with strict latency and data privacy requirements. The challenges and novel research directions arising from the transition towards the edge are summarised, including the development of compression techniques aimed at reducing model complexity and inference time with minimised accuracy losses, as well as the design of compact, low-power hardware and associated software for network deployment. The standard SDLC is expanded to include the emergent set of good practices of ML development, deployment to the edge and maintenance, and encapsulated within the Edge AI Lifecycle. Divided into a pre-development, development, and production stage, common pitfalls and good practices are outlined, with the goal of pushing towards a well-established pipeline and taxonomy in the field of Edge AI. The Pre-Development section summarises the processes of dataset assembly, and problem definition with the translation of the task into one of the ML paradigms. Following that, the Development section addresses the emergent automation of the network design phase, as introduced by evolutionary hyper-parameter search algorithms, and NAS-based methodologies. The section goes on to describe model validation and evaluation tactics, common to all ML applications. The specifics of edge use-cases are then addressed by a categorisation of model compression techniques, and an overview of available edge hardware. Finally, the Production section details a collection of frameworks used for the deployment of optimised models on dedicated hardware and outlines the importance of production monitoring and continuous learning pipelines.

Acknowledgements

This research was conducted as part of the EdgeAI "EdgeAI Technologies for Optimised Performance Embedded Processing" project, which has received funding from KDT JU under grant agreement No 101097300. The KDT JU receives support from the European Union's Horizon Europe research and innovation program and Austria, Belgium, France, Greece, Italy, Latvia, Luxembourg, Netherlands, and Norway.

References

[1] R. Singh and S. S. Gill, "Edge AI: A survey," *Internet of Things and Cyber-Physical Systems*, vol. 3, pp. 71–92, Jan. 2023, doi: 10.1016/J.IOTCPS.2023.02.004.

[2] N. Kukreja *et al.*, "Training on the Edge: The why and the how," *Proceedings - 2019 IEEE 33rd International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2019*, pp. 899–903, Feb. 2019, doi: 10.1109/IPDPSW.2019.00148.

[3] “AI Edge Computing Market Statistics | Industry Forecast - 2030.” https://www.alliedmarketresearch.com/ai-edge-computing-market-A14885 (accessedAug.22,2023).

[4] S. Yi, Z. Hao, Z. Qin, and Q. Li, “Fog computing: Platform and applications,” *Proceedings - 3rd Workshop on Hot Topics in Web Systems and Technologies, HotWeb 2015*, pp. 73–78, Jan. 2016, doi: 10.1109/HOTWEB.2015.22.

[5] T. Sipola, J. Alatalo, T. Kokkonen, and M. Rantonen, “Artificial Intelligence in the IoT Era: A Review of Edge AI Hardware and Software,” *Conference of Open Innovation Association, FRUCT*, vol. 2022-April, pp. 320–331, 2022, doi: 10.23919/FRUCT54823.2022.9770931.

[6] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag, “What is the State of Neural Network Pruning?,” Mar. 2020, Accessed: Aug. 22, 2023. [Online]. Available: https://arxiv.org/abs/2003.03033v1

[7] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations,” *Journal of Machine Learning Research*, vol. 18, pp. 1–30, 2018.

[8] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, “Learning under Concept Drift: A Review,” *IEEE Trans Knowl Data Eng*, vol. 31, no. 12, pp. 2346–2363, Apr. 2020, doi: 10.1109/TKDE.2018.2876857.

[9] C. Shorten and T. M. Khoshgoftaar, “A survey on Image Data Augmentation for Deep Learning,” *J Big Data*, vol. 6, no. 1, pp. 1–48, Dec. 2019, doi: 10.1186/S40537-019-0197-0/FIGURES/33.

[10] M. Jaderberg *et al.*, “Population Based Training of Neural Networks,” Nov. 2017, Accessed: Aug. 22, 2023. [Online]. Available: https://arxiv.org/abs/1711.09846v2

[11] H. Liu, K. Simonyan, and Y. Yang, “DARTS: Differentiable Architecture Search,” *7th International Conference on Learning Representations, ICLR 2019*, Jun. 2018, Accessed: Sep. 07, 2023. [Online]. Available: https://arxiv.org/abs/1806.09055v2

[12] D. J. Pagliari, M. Risso, B. A. Motetti, and A. Burrello, “PLiNIO: A User-Friendly Library of Gradient-based Methods for Complexity-aware DNN Optimization,” Jul. 2023, Accessed: Sep. 07, 2023. [Online]. Available: https://arxiv.org/abs/2307.09488v1

[13] N. Srivastava, G. Hinton, A. Krizhevsky, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[14] M. A. Ganaie, M. Hu, A. K. Malik, M. Tanveer, and P. N. Suganthan, "Ensemble deep learning: A review," *Eng Appl Artif Intell*, vol. 115, p. 105151, Oct. 2022, doi: 10.1016/J.ENGAPPAI.2022.105151.

[15] R. Banner, Y. Nahshan, and D. Soudry, "Post-training 4-bit quantization of convolution networks for rapid-deployment," *Adv Neural Inf Process Syst*, vol. 32, Oct. 2018, Accessed: Sep. 07, 2023. [Online]. Available: https://arxiv.org/abs/1810.05723v3

[16] B. Jacob *et al.*, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, Dec. 2017, doi: 10.1109/CVPR.2018.00286.

[17] Z. Dong, Z. Yao, A. Gholami, M. Mahoney, and K. Keutzer, "HAWQ: Hessian AWare Quantization of Neural Networks with Mixed-Precision," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2019-October, pp. 293–302, Apr. 2019, doi: 10.1109/ICCV.2019.00038.

[18] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: Hardware-Aware Automated Quantization with Mixed Precision," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2019-June, pp. 8604–8612, Nov. 2018, doi: 10.1109/CVPR.2019.00881.

[19] A. Pullini, D. Rossi, I. Loi, G. Tagliavini, and L. Benini, "Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing," *IEEE J Solid-State Circuits*, vol. 54, no. 7, pp. 1970–1981, Jul. 2019, doi: 10.1109/JSSC.2019.2912307.

[20] D. Palossi, A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza, and L. Benini, "A 64mW DNN-based Visual Navigation Engine for Autonomous Nano-Drones," *IEEE Internet Things J*, vol. 6, no. 5, pp. 8357–8371, May 2018, doi: 10.1109/JIOT.2019.2917066.

[21] "SAE J3016 automated-driving graphic." https://www.sae.org/news/2019/01/sae-updates-j3016-automated-driving-graphic(accessedSep. 01,2023).