9

Recent Trends in Edge AI: Efficient Design, Training and Deployment of Machine Learning Models

Mark Deutel[1*], Maen Mallah[2*], Julio Wissing[2*], and Stephan Scheele[3]

[1]Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
[2]Fraunhofer IIS, Fraunhofer Institute for Integrated Circuits, Germany
[3]Ostbayerische Technische Hochschule Regensburg, Germany
[*]These authors contributed equally to this work.

Abstract

With a rising demand for ubiquitous smart systems, processing and interpreting large quantities of data generated on the edge at a high velocity is becoming an increasingly important challenge. Machine learning (ML) models such as Deep Neural Networks (DNNs) are an essential tool of today's artificial intelligence due to their ability to make accurate predictions given complex tasks and environments. However, Deep Learning is computationally complex and energy intensive. This seems to contradict the characteristics of many edge devices, which have only limited memory, computational resources, and energy budget available. To overcome this challenge, an efficient ML model design is crucial that incorporates available optimization techniques from hardware, software, and methodological perspective to enable energy-efficient deployment and operation on the edge. This work comprehensively summarizes recent techniques for training, optimizing, and deploying ML models targeting edge devices. We discuss different strategies for finding deployable ML models, scalable DNN architectures, neural architecture search, and multi-objective optimization approaches, to enable feasible trade-offs considering available resources and latency. Furthermore, we give insight into DNN compression methods such

as quantization and pruning. We conclude by investigating different forms of cascaded processing, from simple multi-level approaches to highly branched compute graphs and early-exit DNNs.

Keywords TinyML, energy efficient AI, neural architecture search, pruning, quantization, cascaded processing.

9.1 Introduction

With the rise of the Internet of Things (IoT), the need to interpret large quantities of data from embedded sensor systems has become ubiquitous. The possible application domains are nearly endless, ranging from Smart Cities over sustainable resource use to agriculture and many more. Yet, including computationally complex data processing with models like Deep Neural Networks (DNN) seems contradictory to the characteristics of small, embedded devices, which usually have substantial limitations regarding energy usage and computing capabilities. To overcome this challenge, multiple approaches have been taken to reduce the energy footprint of machine-learning models, often referred to as TinyML. This work will survey recent literature concerning the most prominent aspects in this field with a focus on embedded sensor systems, which we define as:

- Processing element embedded in a device made for a specific application
- Computations are done by a microprocessor, microcontroller, or FPGA
- Optionally includes an acceleration unit (NPU, DSP, etc.)
- Reduced or no operating system
- Battery or energy-limited

The study starts with scalable DNN architectures in Section 2 Scalable Deep Neural Architecture search to find efficient network topologies in Section 3. The review DNN compression methods to optimize neural networks during and after their training are reviewed e.g., Pruning in Section 4 and Quantization in Section 5. The survey is concluded by looking at cascaded systems in Section 6, which allow partial execution of machine-learning models. Section 7 provides a short summary and discussion.

9.2 Scalable Deep Neural Network Architectures

Scaling DNNs, i.e., increasing their capacity, is a commonly used technique to control the trade-off between the performance of a DNN, e.g. accuracy,

and its resource requirements. The most common form of scaling is achieved either by adding more layers, changing the resolution of the input, or changing the network's with, e.g. by increasing the number of filters in a DNNs convolutional layer, see Figure 9.1. In the following we provide an overview of relevant scalable DNN architectures commonly used on edge platforms.

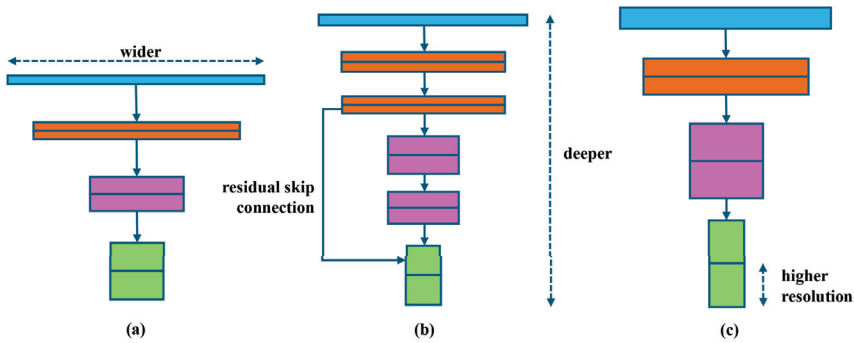

Figure 9.1 Illustration of commonly used approaches for DNN scaling. DNNs can be scaled by either (a) widening the input of the DNN, (b) deepening the DNN by adding more layers and residual skip connections, or (c) increasing the resolution of the feature maps by adding more filters.

9.2.1 Residual networks

Residual networks [1] have been proposed to facilitate the training of very deep neural network architecture. Training such architecture is often hindered by problems such as the vanishing/exploding gradient problem or degradation problems caused by the depth of the DNN [2]. ResNet attempts to address these problems by explicitly fitting layers to a residual mapping instead of the entire underlying mapping. The authors hypothesize that it is easier to optimize the residual mapping than the original unreferenced mapping. They argue that "to the extreme, if an identity mapping were optimal, it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers". In feedforward networks, residual mappings can be realized by introducing "shortcut connections," which are connections that bypass one or more layers. In the case of ResNet, the shortcut connection simply performs an identity mapping and is then added to the result produced by the stacked layers it bypasses. Therefore, introducing it adds almost no computational complexity (just an elementwise added operation).

The standard "residual block" used by the authors in their proposed ResNet architectures consists of two fully convolutional layers and a bypass

connection. However, given the exploding training times for very deep neural networks, the authors also propose an alternative "bottleneck" block. It consists of three convolutional layers (instead of two), where the first and the last are 1×1 pointwise convolutions, leaving the intermediate layer as a bottleneck with smaller input/output dimensions.

9.2.2 MobileNet

MobileNet [3] is a class of DNNs that specifically focuses on deployment on edge platforms, both from a size and runtime perspective. One of the main features of the architecture is the heavy use of depth-separable convolutions, "which is a form of factorized convolutions which factorize a standard convolution into a depth wise convolution and a 1x1 convolution called a pointwise convolution". This means that instead of performing both filtering and combining inputs into a new set of outputs as one operator (standard convolution), the operation is factorized into a separate layer for filtering and a separate layer for combining. The main advantage of this approach is that the factorized representation drastically reduces both the computational complexity and the size of a DNN model compared to using standard convolutions.

To enable scalability, the authors introduce a width multiplier parameter. The role of the parameter is to thin a network uniformly at each layer. Furthermore, they introduce a depth scalar parameter, which is multiplied by the spatial dimensions of the initial input of the network and as a result also scales down the activation tensors of all subsequent layers. In recent years, the authors of the original MobileNet paper have presented two updated versions of the MobileNet architectures. MobileNetV2 [4] is very similar to its predecessors, except that it uses "inverted residual blocks with bottlenecking features". As in the original MobileNet architectures, these blocks consist of a depth-separable convolution but also include a third linear 1x1 convolutional layer that is not followed by any nonlinearity, e.g., ReLU. The authors claim that "experimental evidence suggests that using linear layers is crucial as it prevents nonlinearities from destroying too much information". In addition, the blocks introduce a shortcut connection like that of residual blocks. However, the authors explain that while in regular residual blocks the expansion layers, i.e. the high number of channels, are connected by the shortcut, here the bottleneck layers, i.e. the low number of channels, are connected. Hence the name "inverted residual block". The advantage of this inversion is a much more memory efficient design.

MobileNetV3 [5] improves on its predecessors by introducing lightweight attention modules based on squeeze and excitation networks [6] in the bottleneck structure. Furthermore, all layers are upgraded with a modified version of the swish nonlinearity (h-swish), which the authors claim is faster to compute and more quantization-friendly. Both the swish operator and the squeeze and excitation layers use the sigmoid operator. Since this can be expensive to compute on some platforms, it is replaced by the hard sigmoid (piecewise linear approximation of the sigmoid function). In addition, particularly expensive layers of MobileNetV2 have been redesigned to be more efficient. This includes the last layers of the previous architecture, where the authors describe that they were able to „drop three expensive layers at the end of the network at no loss of accuracy."

9.2.3 EfficientNet

EfficientNet [7] is a class of eight differently scaled DNN architectures (B0-B7). The scaling is performed as a combination of depth, width, and resolution scaling, which the authors call compound scaling, and which they control by an additional parameter φ. The authors note that increasing the three available scaling dimensions comes at different costs which means that a trade-off between the parameters must be made. For example, the authors mention that for regular convolutional operations, doubling the network depth parameter will double the FLOPS, but doubling the network width parameter or resolution parameter will even quadruple the FLOPS.

As their baseline network, the authors use a regular CNN (B0) targeting 400M FLOPS, which the authors searched for by using a multi-objective neural architecture search that optimizes both accuracy and FLOPS, and which they based on their previous work [8]. The authors used $ACC\,(m) * \left(\frac{FLOPS(m)}{T}\right)^{w}$ as their optimization objective, where $ACC\,(m)$ and $FLOPS(m)$ denote the accuracy and FLOPS of the model while m and T denotes the target FLOPS, and w $= -0.007$ is an additional hyperparameter used to control the trade-off between the two metrics. From their baseline network (B0), the authors then use a two-step process to derive all seven other architectures (B1-B7):

- *STEP 1*: assuming twice as many resources are available as a starting point and then do a small grid search of the three scaling parameters for depth, width, and resolution.

- *STEP 2*: fix the best-found depth, width, and resolution parameters as constants and scale the baseline mesh with different φ using to get the final set of architectural trade-offs B1-B7.

9.2.4 Scalable weights

Another common approach to achieve dynamic scalability at runtime is to adapt trainable parameters ("weights") based on the received input. A conditionally parametrized convolutional layer (CondConv) is proposed in [9] that can be used as a drop-in replacement for standard convolutional layers. Their design parametrizes the kernels of a regular convolution as a linear combination of n experts $W_1, \ldots, W_n$ weighted by $\alpha_1, \ldots, \alpha_n$ functions of the input learned by gradient descent. The authors argue that "this is much more computationally efficient than increasing the size of the convolutional kernel itself, because the convolutional kernel is applied at many different positions within the input, while the experts are combined only once per input". In conclusion, CondConv can introduce the same amount of additional capacity as MoE while being more computationally efficient by requiring only one convolution.

Deformable convolutions [10], [11] implement the concept of dynamic receptive fields by sampling feature pixels during the computation of convolutions from adaptive locations based on the input. The offset of each sample is generated using an additional convolution that generates an offset field based on the input feature map. As a result, deformable convolutions generalize various transformations for scale, (anisotropic) aspect ratio and rotation. The concept of weight prediction, first proposed by [12] allows for directly modifiable weights in a feedforward network that are contextually modified at runtime by a second controller network. The authors propose this architecture as a memory efficient alternative for representing temporal information compared to recurrent neural networks. The idea was adapted to more modern DNN architectures by [13], [14]. Here, the filters of a convolution are dynamically generated from the input using an additional "filter generating network". The two inputs of the convolution can be either identical or different, depending on the task at hand.

9.2.5 Practical Considerations

Scalable DNN architecture helps to easily adapt a network to a given set of resource constraints, often by simply tuning a set of hyperparameters that

do not require extensive prior knowledge about the internal structure of the DNN. This makes the architecture presented in this section ideal candidates for deployment on edge systems or other resource constraint environments. In addition, scalable DNN architectures are versatile in that they can easily be used in combination with other techniques such as neural architecture search, pruning, and quantization, which we will discuss later in this review.

9.3 Neural Architecture Search for Resource Aware DNN Deployment

The search for a feasible DNN architecture that can be deployed on an edge system with given resource constraints is typically formulated as a multi-objective hyperparameter optimization (HPO) problem [15], solving a black-box function that maps from a set of DNN architecture-specific hyperparameters to a set of target metrics such as accuracy, memory consumption, latency, or throughput, on which platform-specific constraints are defined. In this definition, sampling the black box is equivalent to training a DNN with a specific configuration on a given dataset. Often synonymously used with HPO is Neural Architecture Search (NAS). NAS describes the process of finding good DNN architectures in an automated, non-human controlled way. Although it is primarily concerned with only maximizing DNN performance, i.e., solving a single-objective optimization problem, most concepts can be intuitively extended to a resource-aware multi-objective search. As a result, multi-objective NAS for edge platforms has recently become more of a focus as well, e.g. [16, 8]. There are three approaches to NAS that are prominently discussed in the literature: First, black-box HPO [17]-[20], second, differentiable NAS [21], [22] and third, zero-cost NAS [16], [23], [25]-[30]. Black-box HPO works reliably and can be easily extended to the multi-objective case, but it is also slow and often inefficient sample, requiring many different DNNs to be trained and evaluated.

Differentiable NAS relaxes the optimization problem so that the architecture can be optimized as part of the regular training of a DNN. However, recent research suggests that differentiable NAS has stability problems and does not generalize well [24]. Finally, zero-cost NAS is very time-efficient since it does not train DNNs directly but uses an empirical surrogate model. As a result, it does not provide accurate information about the performance of an architecture, but only simple statistics derived from the surrogates [31]. In the following, we discuss each of the three approaches and their implications for resource-aware NAS.

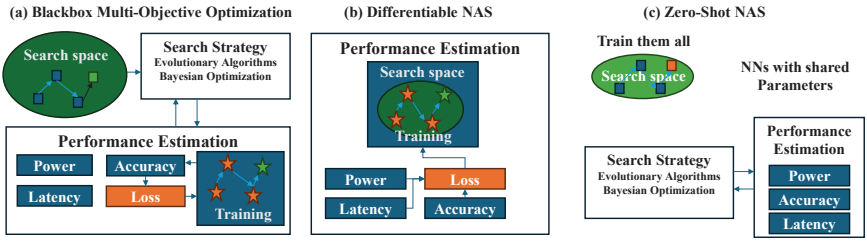

Figure 9.2 Three types of NAS are considered in the literature: (a) For black-box multi-objective optimization, many DNNs must be trained. However, optimisation results in a Pareto set of exact trade-offs between the different objectives. (b) Differentiable NAS returns only a single trade-off but is time and resource efficient because it optimizes the DNN during a single training run. (c) Zero-shot NAS allows fast DNN specialization for deployment goals, but the performance of the proposed trade-offs is only estimated.

9.3.1 Black-Box Multi-Objective optimization

The most common approach to solving HPO problems is black-box optimization. A traditional approach to tackle black-box (multi-objective) optimization is evolutionary algorithms (MOEA), one of the most common being NSGA-II [32]. As an example, [33] explore hyperparameter and compression parameter optimization using evolutionary algorithms for DNN deployment and combine it with pruning and quantization. A drawback of MOEA is that due to their population-based nature, they are sample inefficient, which can be time and resource consuming in the case of NAS, where many DNNs need to be trained and evaluated.

To overcome this problem, Bayesian black-box optimization [34] can be considered as a more sample-efficient alternative to MOEAs. Here, cheap to evaluate surrogate models, usually Gaussian processes (GPs), are used to approximate the black-box objective functions. During optimization, for each trial performed, a nested optimization is performed on the GPs, which have been previously fitted with an acquisition function based on all trials observed so far. The result of this nested optimization is in turn used to propose the next parameterization (or set of parameterizations) to the outer optimization loop, which in turn evaluates them on the actual objective functions, thus repeating the iterative optimization process.

Black-box optimization for HPO has also been regularly combined with reinforcement learning (RL). For example, [17] observed that a DNN architecture can be defined by a variable-length string, and that it is therefore possible to use a recurrent neural network (RNN) to generate such strings.

The authors call such an RNN a "controller". By training the DNNs (child networks) generated from the strings emitted by the controller, accuracy on a data set can be obtained, which in turn can be used as a reward signal. This signal can then update the controller (i.e., the controller learns to improve its search over time through RL).

Another approach to neural network design with reinforcement learning is based on using a Q-learning agent that samples a CNN topology conditioned on a predefined behavior distribution and the agent's prior experience [18]. The authors define the layer selection process as a Markov decision process. The agent sequentially selects layers via an ϵ-greedy strategy until it reaches a termination state. Like [17], the reward given to the agent is constructed from the validation accuracy of the generated network.

In addition, the authors use a replay buffer that stores the network topology and predict performance on a validation set for all sample models. This means that if an already trained model is resampled, it is not retrained, but the previously found validation accuracy is presented to the agent.

A major problem with all black-box HPO approaches is that they require many different DNNs to be trained and evaluated, which can become a time and resource consuming task. In addition, black-box optimization struggles with large parameter spaces, as the sampling complexity increases exponentially with each parameter added to the search space, an effect informally referred to as the "curse of dimensionality" [35].

9.3.2 Differentiable NAS

Unlike Black-box HPO, Differentiable NAS [21], [22] does not search over a discrete set of candidates architectures but instead relaxes the search space to be continuous. As a result, the target architecture can be optimized with respect to the performance of the validation/test dataset using gradient descent, solving the problem of having to evaluate many different architectures. Since gradient descent is more data efficient than regular black box search, differentiable NAS can achieve competitive performance to NAS while requiring much less computational resources.

The search space considered by differentiable NAS consists of computational cells as building blocks of the final architecture. Each cell is an acyclic graph containing an ordered sequence of nodes. Each node is a latent representation, e.g. a feature map, and directed edges connecting them are operations that transform the latent representation. This means that each intermediate node is computed based on all its predecessors.

Differentiable NAS can be seen as a bi-level optimization problem. The goal for the architecture search is (a) to find architecture, i.e., a combination of building blocks that minimise computational complexity and (b) a set of trained weights associated with the architecture that minimizes the training loss.

Another variant of differentiable NAS is hierarchical NAS [36], [8]. It not only searches at a cell structure level but also at network structure level, thereby formulating a hierarchical search space. The authors noticed that in modern CNN design the outer network level controls spatial resolution changes, while the inner cell level controls specific layer-wise computations. Regular NAS follows this principle as well but only automatically searches the inner cell level while the outer network level is designed by hand. This can become problematic for use cases which are sensitive to spatial resolution changes.

9.3.3 Zero-Cost neural architecture search

Like differential NAS, zero-cost NAS attempts to minimize the search costs associated with black-box optimization. However, unlike differential NAS, it focuses on a redundant learning strategy to minimize computational cost by using a well-trained weight-sharing model, i.e., a supernet from which many architecture variants, i.e., subnets, can be drawn at zero-cost. To find the best architecture α^* under given resource constraints, a zero-cost proxy metric is used that can score a subnet without training and validation on a dataset.

Several proxies have been discussed in research: [26] use the overlap of activations between data points in untrained networks, [27] quantify the expressiveness of an architecture, which they call Zen-Score, while other works used gradient-based zero-cost proxies such as the gradient norm to estimate model performance [28–20].

In recent research, zero-cost NAS approaches have been proposed to easily allow for specialization of network architectures to different target platforms. This allows for an efficient search of architectural variants for differently scaled edge systems.

For example, once-for-all∼networks [16] provide an optimization strategy tailored to a multi-objective search space with different device and resource constraints. To achieve this, the authors decouple network training and search. During the training phase, the authors train a single network using a technique they call *Progressive Shrinking*. Once the single network is

trained, different submodels are sampled from it and evaluated, requiring no further (re)training. These subnetworks can have different input resolutions, different kernel sizes or filters in their convolutional layers, and a different number of layers.

Another example of network specialization using zero-cost NAS is PreNAS [25], where the authors improve the efficiency of regular black-box optimization (which the authors call "one-shot NAS") by combining it with zero-cost NAS. To improve search efficiency, the authors propose to construct a reduced ("preferred") search space based on high-quality architectures selected under various resource constraints using zero-cost NAS, which is easier to search by black-box optimization than the original "full" search space.

9.3.4 Practical considerations

One of the biggest challenges that must be considered when using NAS as part of DNN deployment for embedded targets is the typically large amount of time required by the techniques we presented in this section. This is especially true for the black-box optimization which, while generally providing consistently good results, requires extensive sampling of the underlying search space which is the most expensive part in NAS. Other techniques presented in this chapter, namely Differential NAS and Zero-Cost NAS attempt to alleviate this problem, either by relaxing the problem to be differentiable so that it can be solved as part of network training or by relying on cheap-to-sample, often model-based surrogates of the expensive to sample search space. However, both techniques have caveats that should be considered: Differential NAS has been shown to have robust problems with optimal architectures often found to generalize poorly [24]. Furthermore, Differential NAS does not produce a Pareto front from which an optimal trade-off can be selected, but only a single architecture that is considered optimal after training. Many Zero-Cost NAS techniques, while allowing extremely fast DNN specialization for different deployment targets and resource constraints, report only simple statistics about found architecture candidates, not actually trained and evaluated networks. To summarize this section, choosing the best NAS strategy in the search for DNN architectures feasible for edge deployment is a trade-off between search time and the quality of the result obtained. For the best possible results, black-box optimization, ideally combined with Bayesian optimization, should be chosen, while for fast but less accurate results, zero-cost NAS techniques should be preferred.

9.4 Deep Neural Network Pruning

A common way to achieve compression of DNN to allow its use in resource-constrained environments is DNN pruning. Pruning is based on the idea that some of the trained parameters of a DNN can be removed, i.e. pruned, without significantly degrading the accuracy of the network. It is based on the understanding that most neural networks are overparameterized and have redundancy in their trained weights [37]. The approach is well known and has been discussed by several authors as early as the late 1980s [38]-[40].

While at first pruning was mainly proposed with better network generalization, less overfitting, and improved learning speed in mind, today it has become a popular technique to achieve DNN compression with minimal or no loss of accuracy.

The simplest way to prune a DNN is to set a subset of its trainable parameters to zero during training resulting in sparse parameter tensors. By setting parameters to zero, they are removed from the scope of the optimizer used to train the DNN. Therefore, the removed parameters no longer affect the training and validation loss of the network.

How pruning is performed during training can be roughly characterized by a few different parameters, which we will discuss in the remainder of this section.

9.4.1 Pruning granularity

There are two common techniques to perform DNN pruning: *Unstructured Pruning* and *Structured Pruning*. Their main difference is the granularity at which the techniques introduce sparsity into a DNN.

Unstructured Pruning introduces sparsity by removing individual neurons from the parameter tensors of a DNN while structured pruning removes entire structures of neurons. Typically, Structured Pruning targets structures such as filters [41] or channels [42] in convolutional layers. However, it is possible to extend Structured Pruning to rows or columns of linear layers as well.

An advantage of Structured Pruning over Element Pruning in the context of DNN compression, is that pruned structures can not only be set to zero but can be completely removed from the parameter tensor. This is not possible with element pruning, which produces matrices of arbitrary sparsity. However, Structured Pruning also has drawbacks: Removing structures from a DNN is much more invasive than Element Pruning, and its implementation is not as simple as setting individual values to zero. In regular feed-forward DNNs, layers are usually connected in a sequence. Information is passed

through the network from top to bottom. Therefore, the output feature map of one layer becomes the input feature map of the next. By changing the shapes of the parameter tensors in these layers, the shapes of their input and output feature maps change as well. Therefore, removing structures from layers inherently means removing the data dependencies between them. This makes the implementation of Structured Pruning complex and requires a global view of the DNN structure. Structured Pruning becomes even more complicated in branching feed-forward networks, such as residual networks [1]. Here, data dependencies can span multiple layers run in parallel instead of just sequential layers.

It is possible to apply Structured Pruning and Element Pruning to a DNN together. [43] describe how they first apply Structured Pruning to convolutional layers of a DNN and then apply Element Pruning to the remaining structures. They call this approach *Hybrid Pruning*. They argue that Structured Pruning is a coarse-grained method, while Element Pruning is a fine-grained one. Therefore, Element Pruning can be used to remove connections that would otherwise be missed by Structured Pruning. In addition, Element Pruning and Structured Pruning can be used together by applying them to different layers or layer types in a DNN.

9.4.2 Pruning heuristics and sensitivity analysis

A major challenge in DNN pruning is to decide which elements or structures have the least impact on the validation loss of a DNN when removed.

The most accurate approach to achieving optimal pruning would be to remove each structure or element of a network one by one and evaluate its impact on the loss. This is referred to as the oracle criterion [44]. However, the downside of this strategy is that it is extremely resource and time consuming. So much so that it is inapplicable in most scenarios.

Therefore, other heuristics have emerged to approximate optimal pruning in a more computationally efficient manner. The process of quantifying the importance of parameter tensors of DNNs is referred to as *Sensitivity Analysis* [45]. Much research has been done in recent years to find good approximations for both Element and Structured Pruning techniques. In the following, we give an overview of some of the most used heuristics.

9.4.3 Magnitude or threshold based heuristics

Early work focused heavily on second-order derivative-based heuristics to approximate pruning. For example, [39] and later [40] propose calculating

saliency scores for elements to determine their usefulness. They then use these scores to zero out a certain number of elements that they consider to be the least useful. [39] call this approach *optimal brain damage* (OBD). To avoid having to compute the saliency using the oracle approach, their heuristic approximates the objective function of a neural network using a Taylor series. However, this method requires additional computation.

As a result, more recent work has focused on simpler heuristics to approximate the usefulness of the trained parameters. The most common ones use simple threshold functions [45], [46] They consider a parameter useful if its absolute value is above a certain fixed value. If instead the value of the parameter is below or equal to the fixed value, it is marked as a candidate for removal. The heuristics are generally based on the understanding that a higher value creates a greater activation and is therefore more likely to have a significant impact on a layer's output than a smaller value. However, manually defining good thresholds requires extensive network analysis and is not intuitive. To solve this problem, most of the authors mentioned above approximate the distribution of values in the weight tensors of neural networks with a Gaussian distribution with a mean of zero. They then use the standard deviation of the distribution in combination with a scaling factor to automatically define pruning thresholds on a per-tensor basis. As a result, the only unknown variables remaining are the scaling factors. [45] derive these factors by increasing the sparsity in the layers of an unpruned baseline network and monitoring its accuracy changes. However, this is expensive because the baseline network must be trained first.

9.4.3.1 L-Norm heuristics

Threshold and magnitude-based approaches are not only useful for evaluating the importance of single parameters, but also for ranking complete parameter structures of neural networks. Therefore, a common way to evaluate the importance of such structures is to sort them by their l1-norm. [41] describe how they use the norm to estimate the overall size of filters in convolutional layers. Furthermore, using the norm also gives an estimate of how large the values in the resulting output feature maps will be. In addition, the l2-norm is sometimes considered as a means of classifying parameter structures in neural networks.

9.4.3.2 Gradient Ranked Heuristics

Another heuristic for structure pruning has been proposed by [44]. The authors present a parameter structure ordering heuristic based on Taylor series

derived from the difference in loss when certain parameters are removed from a network. The approach is like the OBD heuristic described earlier in this section. This is also explicitly stated by the authors. Their resulting heuristic accumulates the product of the activation tensors and the gradients of the cost function with respect to the activation tensors. The gradients can be easily obtained via the backpropagation algorithm. Intuitively, the heuristic considers structures to be less significant if their parameters are close to zero and have a flat gradient.

9.4.3.3 Activation based heuristics

Another approach to approximate the importance of parameter structures is proposed by [47]. The authors analyze the sparsity of activation tensors during training. They do this by monitoring the average percentage of zero values (APoZ) found in outputs computed by rectified linear unit (ReLU) activation functions throughout the network. The higher the percentage of zero values generated by the ReLU, the less significant the structure is considered by the heuristic. [44] also mention the ApoZ approach in their paper. However, the authors caution that the approach may not perform well on early layers of neural networks. They note that early layers are typically trained to detect less defined features, resulting in denser activation tensors. Only for later layers, when feature detection has become more precise, do the activation tensors become sparse.

9.4.3.4 Relevance-based heuristics

Another criterion for pruning neural network structures is based on Layer-wise Relevance Propagation (LRP) [48] originating from the field of explainable AI (XAI). This approach assigns relevance scores to individual neurons in a neural network. Traditionally LRP serves as an XAI method to highlight the relevant parts of a given input to generate a local explanation for interpreting complex non-linear machine learning models, e.g. the relevancy of pixels of an input image can be highlighted in terms of a heat-map representing the influence of input parameters that are decisive for an image classification.

This relevance-based pruning criterion for pruning CNNs works by iteratively pruning the least relevant units, i.e. weights or filters of a network. The relevance scores are obtained via back-propagation from the output layers to the input layers and are assigned to all neurons of all layers, where the main characteristic of LRP is the backward pass through a network that can be computed efficiently. The relevance scores are then used to identify the least relevant units, which are the ones that can be pruned to discard all

aspects of a network that do not contribute to the decision of a model. A pruned network can then be fine-tuned to recover possibly lost accuracy. This process is repeated until the desired level of compression is achieved. The LRP criterion is shown to be effective in reducing computational and parameter costs. Furthermore, based on experiments, the authors in [49] show that this pruning criterion performs consistently well or even better than state-of-the-art pruning criteria when model refinement and fine-tuning is applied.

9.4.4 Pruning schedule

A pruning schedule, sometimes called a pruning recipe, describes when, how often, and how much of a network is pruned during training. A very straightforward approach to schedule pruning is referred to as *one-shot pruning*. Its use has been described in early papers such as [39]. The general idea is to first train a network until it reaches a reasonable accuracy. Then, the whole network is pruned using a certain heuristic to remove the structures or elements with the lowest score. Based on this score, several of them are removed. Additionally, the authors found it beneficial to retrain the network after pruning.

One-shot pruning schemes are also described in more recent papers, for example by [44, 45]. However, both authors also distinguish another type of pruning schedule. They call it the *iterative pruning schedule*. This approach focuses very strongly on the idea of pruning and retraining a network several times instead of just once. Therefore, not all parameters are removed at once, but step by step over several pruning iterations as part of training. This allows the network to adapt more gradually to the decreasing set of trainable parameters.

Both authors also suggest specifying the number of trainable parameters to be pruned on a layer-by-layer basis rather than for the entire network. This is based on the understanding that the layers of a neural network may have different sensitivities to pruning. [44] show that some layers can be pruned much more aggressively than others before any degradation in the accuracy of a trained model becomes noticeable.

An extension to iterative pruning schedules is presented by [50]. The authors build on the idea that the number should be gradually increased instead of removing a constant number of parameters in each pruning iteration. They propose an algorithm that automatically increases the number of pruned parameters in a DNN over a range of n pruning steps based on a

few predefined parameters. They call this the *Automated Gradual Pruning* algorithm. They state that the intuition behind the algorithm is "to prune the network rapidly in the initial phase, when the redundant connections are abundant, and to gradually reduce the number of weights pruned each time as fewer and fewer remain in the network".

9.4.5 Practical considerations

Pruning is a core technique when it comes to fine-tuning the size of a DNN for a given problem, i.e. dataset, during training. The technique has proven to be extremely versatile, even when combined with other compression techniques such as quantization or NAS. To conclude this section, we would like to point out some practical considerations when using pruning as part of a DNN deployment pipeline: First, structured pruning will lead to immediate performance gains because pruned structures can be transparently removed from tensors, resulting in fewer computations, without any changes to the environment used to run the DNN at runtime. Unstructured pruning, on the other hand, only creates sparse tensors that require support from the environment to be executed efficiently at runtime. Second, the choice of a reasonable pruning schedule is of great importance, with iterative pruning using Automated Gradual Pruning generally providing better results. Third, simpler heuristics such as L-normed-based heuristics offer a good trade-off between accuracy and speed, while LRP based heuristics can provide more informed decisions and tend to produce an overall better pruning result at the cost of often higher compute times. Still, the outcome of pruning using LRP depends highly on the network and layer types used, and multiple parameters need tuning to make it applicable in the general case, see [51].

9.5 Quantization

One of the main methods to reduce the energy consumption of DNNs is to reduce their size and computational complexity using quantization [52]. Quantization of DNNs refers to the quantization of both the parameters (weights, biases, scaling factor ...etc.) and activations. DNN Quantization corresponds to some of or all the following benefits (depending on the model of quantization):

- Reduction of the NNs' size which reduces the required energy to transmit and store the trained models when deployed on edge.

- Reduction in the complexity of the operations (mainly Multiply and accumulate MAC) leading to reduced power consumption of these operations,
- Reduction in size (bit-width) of the NN parameters and activations leads to improved power efficiency regarding memory access, required buffers and moving the data around on the edge target device.

9.5.1 Quantizers

First, we would like to define what quantization is, its different types and all the relative terms. The mathematical operation maps the input space into quantified values in the output space and can be divided, in general, into uniform and non-uniform. As the name suggests, non-uniform quantization produces non-uniform output values which can be represented by a floating-point and achieved using lookup tables. This is part of the model compression which will not be covered in this section, as it helps to reduce the model size for transmission or storage purposes but does lead to computational benefits or reduction in energy consumption.

On the other hand, uniform quantization is the most common method used in DNN quantization. Here, the input space is mapped uniformly to the output space according to the following generic equation.

$$\mathrm{x_Q} = \mathrm{clamp}\left(\frac{\mathrm{round(x)}}{\mathrm{s}} + \mathrm{z}, \mathrm{a}, \mathrm{b}\right) \ , \tag{9.1}$$

where $round()$ is a generic rounding function, s is a scaling factor, z is a zero-point, a and b are the minimum and maximum that define the range, and $clamp(x,\ a,\ b)$ is a clipping function defined as follows:

$$clamp(x, a, b) = \left\{ \begin{array}{ll} a & if\ \ x \leq a \\ x & if\ \ a < x < b \\ b & otherwise \end{array} \right\} = min(max(x, a), b) \ . \tag{9.2}$$

The implemented Rounding has an important impact on the quantization procedure and results. Here we list some of the commonly user rounding schemes:

- **Nearest**: $round(x) = \lfloor x \rceil$.
- **Round down**: $round\,(x) = \lfloor x \rfloor$.
- **Round up**: $round\,(x) = \lceil x \rceil$.
- **Stochastic rounding** [53]: stochastic rounding rounds the values up or down stochastically, $round\,(x) = \lfloor x \rfloor + (p > x - \lfloor x \rfloor)$ where p is a random variable sampled from a uniform distribution.

- **Adaptive rounding** [54]: where each individual value is rounded up or down to minimize the overall loss in the network.

Equation (9.1) defines an affine uniform quantization A.K.A. asymmetric uniform quantization. It can be further restricted by limiting $a = 0$ and $b = 2^{bw} - 1$ to produce integers with bit-width of bw. In addition to the benefits of reducing the operands bit-width and thus the required energy for memory access, the integer operations consume less energy [55] - [57] leading to even further energy savings. Additionally, a symmetric singed-integer quantization can be achieved by restricting $a = -2^{bw-1}$, $b = 2^{bw-1} - 1$ and $z = 0$. Finally, we can restrict this to represent fixed-point arithmetic by limiting s to powers of 2. This restriction helps to reduce energy consumption further as it does not require any multiply or divide operations but rather only bit shifts.

The quantization operation defined in equation (9.1) is irreversible, but an approximation can be achieved with:

$$\mathrm{x} \approx \overline{\mathrm{x}} = \mathrm{s}\left(\mathrm{x_Q} - \mathrm{z}\right) = \mathrm{s}\left(\mathrm{clamp}\left(\frac{\mathrm{round(x)}}{\mathrm{s}}, \mathrm{a}, \mathrm{b}\right) - \mathrm{z}\right) \ . \tag{9.3}$$

The quantization error/noise is then measured using $q_e = x - \overline{x}$. This error affects the correctness of the operations, but it was found that NNs are resilient to such errors and the performance is negligible or can be managed using different methods.

9.5.2 Granularity

After we defined and explained quantization and rounding schemes, we will look at how to apply these to neural networks. The basic building block of NNs is the matrix multiplication A.K.A multiply-accumulate (MAC) defined as $O = \overrightarrow{b} + \boldsymbol{W}\boldsymbol{X}$, where $\overrightarrow{b}$ is the bias vector, $\boldsymbol{W}$ is the weights array, and $\boldsymbol{X}$ are the inputs. This operation is then approximated as follows:

$$\mathrm{O} = \overrightarrow{\mathrm{b}} + \mathbf{W_q}\mathbf{X_q} \ , \tag{9.4}$$

where $\mathbf{W_q}$ and $\mathbf{X_q}$ are the quantized tensors w.r.t $\boldsymbol{W}$ and $\boldsymbol{X}$. It is worth noting that the quantization parameters such as scaling factor, zero-point and range (bit-width) are set separately for the wights and activations. Equation (9.4) quantizes both the weights and activations it is also possible to quantize only the wights [58] – [59]. Weight-only quantization is generally simpler and easier but misses on a lot of the benefits of additionally quantizing the activations as the first only makes use of smaller size and less memory

access but still requires floating-point operations. On the other hand, quantizing activations normally requires a dataset (unlabeled) to determine the appropriate range and scaling factors for each layer. The bias was left out of quantization as it normally required more bits to maintain high accuracy. E.g. the authors in [60] use 8 bits for the weights and activations and 32 bits for the bias.

The quantizers (scale, zero-point, range, bit-width ...etc.) can be specified for each layer (weights and activation) separately, this is referred to as per-layer quantization. Moreover, others have shown that reducing the granularity even further and performing channel-wise quantization (i.e. specifying a different quantizer per output channel/kernel) improves the results [61]–[63]. Others went beyond and quantized per-group of weights or activations [64]–[65]. The trade-of here is often that increasing granularity improves performance and accuracy but comes with an extra overhead in the form of extra parameters per quantizer and thus more memory access and required buffers.

9.5.3 Methods

Here, we discuss the two main categories to quantize NNs with minimal performance degradation and how do they compare. The first approach (Post-Training Quantization PTQ) is simpler as it takes in a trained NN and performs the quantization after training, while the second (Quantization-Aware Training QAT) is more complex as the quantization must be accounted for during training.

9.5.3.1 Post-Training quantization

Early approaches of quantizing NNs relayed on post-training quantization due to its simplicity. In these methods, the NN expensive computations (training) is performed once and then the trained NN can be quantized using different techniques. Using this method, NNs can be quantized to 8-bit weights [62] and [66] or even 4-bit weights [59], [63], [64], [67] and [68] or a mix of the two [69] with zero or minimal degradation in accuracy and performance.

Additional activation quantization is more complicated than weight-only quantization as it produces more errors that propagate though the network. E.g. In [70] the authors use 8-bit and kept the mixed activations with INT8/FP16 due to the high required dynamic range. Moreover, unlike weight quantization, activation quantization often requires training or validation

datasets to collect the statistics about the data to determine the quantization range and scaling required. Yao et al. Managed to quantize the same large language model to 8-bit activations using Token-wise Quantization and knowledge distillation [69]. To determine the activation range, [63] uses the min/max and therefore no clipping occurs, while [63] uses Analytical Clipping for Integer Quantization ACIQ to clip the outliers and achieve a tighter range producing less overall rounding error. In [71], the authors get around the need for training or validation data by using distilled dataset that is designed to match the statistics of the original data. On the other hand, [72] learned a parameterized (range) activation function during training that can be used to clip the values. This approach, however, might not be considered as a post-training quantization approach as it requires changes to the training procedure.

9.5.3.2 Quantization-Aware training

In the previous section, we explained how a trained network can be quantized to 8-bit and even in some cases 4-bit weights and activations with minimal or no loss of accuracy. However, in most cases, quantizing the NNs beyond 8-bits yields significant performance degradation. This degradation can be mitigated if the quantization is accounted for during training. This is commonly referred to as Quantization-Aware Training QAT [60], [73] and [74]. Simply put, the NN graph is updated with the same quantizers from Section 6.1 into the parameters (weights, bias, ...etc.), activations or both during training. This way the training of the NN will account for the quantization error producing more robust NNs.

The biggest question in incorporating the quantization operation into training comes from back-propagation/gradient descent and its reliance on

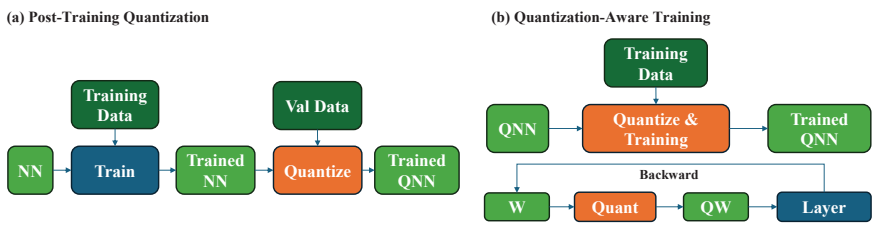

Figure 9.3 The workflow of the two main quantization methods: (a) Post-Training Quantization and (b) Quantization-Aware Training (including the straight throw estimator for the backpropagation)

derivatives. Mainly, the rounding operation in these quantizers has zero derivatives almost everywhere. Therefore, we cannot back propagate the error to update the networks parameters. To solve this question, Begio et al. analyzed and studied four different approximations [75]. Of which, the most promising is the straight-through estimator [76]. This estimator is quite common due to the simplicity as it replaces the derivative with the identity derivative, and it's proven to achieve the task of training.

9.5.4 Practical considerations

To make sure the review on NN quantization is comprehensive, a few practical considerations must be considered by the designers.

- **PTQ vs QAT**: PTQ is generally preferred over QAT when the goal is INT8 or even sometimes INT4 due to simplicity. However, QAT is preferred for sensitive applications or when severe quantization is required.
- **Floating-point vs Fixed-point quantization**: We have covered mostly Fixed-point quantization as it leads to reduction in model size and computational requirements. On the other hand, Floating-point quantization is helpful to reduce the model size for compression purposes but does not lead to a reduction in computation requirements. This is because the model must be uncompressed to the original values for inference.
- **Fusing layers**: The quantization operation can often be fused into the activation layer e.g. ReLU [63] to improve the quantization range as ReLU clips all the negative values anyways. Moreover, it can be fused into the normalization layer [69] and [77] to reduce the overhead cost as both operations require scaling and shifting the values.
- **Layer equalization**: It is observed that quantization to <INt8 often leads to degradation of accuracy, especially in depth-wise convolutions due to the high dynamic range within the layer [78]. One method to compensate for the high dynamic range is to use a channel-wise quantization as mentioned earlier which introduces complexity as each channel will require a different scale and bias parameters. Another solution is to perform cross layer equalization [78], Batch norm tuning [77], bias correction [79] or weight factorization [80].
- **Hyper-parameter selection**: Quantization of NNs and especially QAT adds few more parameters that need to be set/optimized for best performance (power/latency vs accuracy). This is not specific for NN

quantization, but it adds to the complexity for the optimization task. Refer to Chapter 4 for more details.

9.6 Cascaded Processing

In addition to the techniques mentioned above, it is also possible to reduce the energy footprint of a smart sensor system by introducing cascaded processing. Instead of using one big model, this approach relies on multiple models executed one after another (cf. Figure 9.4). By including exit points between models, the system can trigger the execution of the next one lazily, only using the energy needed for a particular decision. Each model in this chain should be designed in a way that builds upon the decision from the previous model(s). In this way, the processing of one model might suffice to conclude so that at every point, the execution may be aborted.

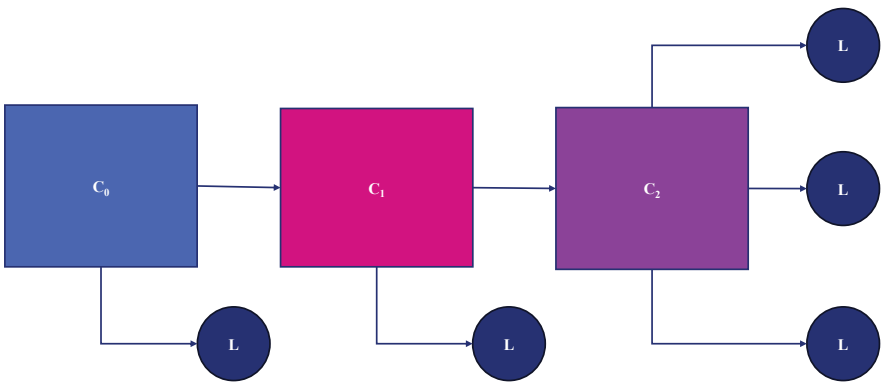

Figure 9.4 Example of simple cascade. Three classifiers are executed one after the other to classify three different labels.

This section will cover the current most relevant literature for cascaded processing, starting with traditional hierarchical models and their applicability to distributed sensor systems. Afterward, we will cover Early-Exit-NNs, which bring the same idea to neural networks.

9.6.1 Hierarchical systems

A typical example of a hierarchical system can be found in condition monitoring as depicted in Figure 9.5: Given a sensor system that should

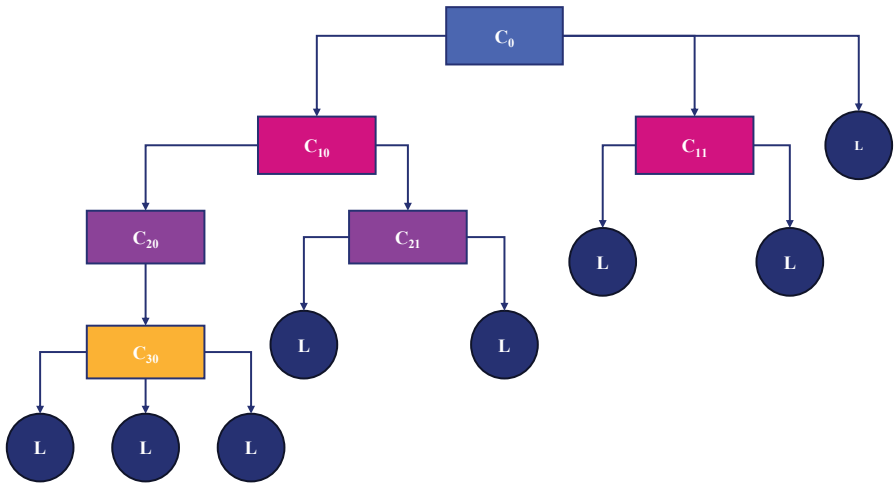

Figure 9.5 Complex hierarchy with multiple levels. The classification flow is separated into different branches.

classify the current machine status into different fault types and severities. The cascade's first model could detect anomalies, classifying abnormal and normal behavior. As this decision can be made quickly, the model should be small and can be run continuously. If a fault has been found, the following model classifies fault types. At this point, the processing can be stopped if the found anomaly has been a false positive. Otherwise, the second model is followed by an additional model for each fault type that classifies the severity of each fault. In that way, the models in the severity level can be specifically tailored towards a distinct fault type, leading to a more efficient model for the individual decision.

Building a foundation for hierarchical models, Silla and Freitas survey different applications for hierarchical classifications in [79]. While the possible application domains are interesting, this paper's most relevant work is how the authors generalize hierarchical classification in a unified framework. First, they introduce a class taxonomy as a poset $(C, \prec)$. C is a finite set of class concepts with a partial order $\prec$ over C, which is asymmetric, antireflexive, and transitive. This definition matches a directed acyclic graph (DAG) but is often simplified to a tree structure, where each node is limited to one parent node.Additionally, the authors cluster models for hierarchical classification problems into four groups: Local classifier per node (LCN), Local classifier per parent node (LCPN), local classifier per level (LCL), and

global classifier (GC). LCN classifiers are found in models with a binary classifier at every node, whereas LCPN models can include multi-class classifiers. The LCL approach uses only one multi-class classifier per level, meaning no branching can be done with LCL classifiers. A GC uses just one big model to classify each label.Based on this definition, the authors categorize a hierarchical classification problem in a 3-tuple (Υ,Ψ,Φ) ,where Υ specifies the type of graph (DAG or Tree), Ψ whether a class label is associated with a single path (SPL) or multiple (MPL), and Φ the depth of labels from full depth (FD) to partial depth (PD). Additionally, they describe a categorization of different types of hierarchical algorithms as a 4-tuple $(\Delta,\Xi,\Omega,\Theta)$, where Δ in association with Ψ represents if the algorithm performs single path prediction (SPP) or multipath prediction (MPP), Ξ refers in line to Φ to mandatory leaf-node prediction (MLNP) or non-mandatory leaf-node-prediction (NMLNP). Ω reflects the model's ability to perform on tree or DAG-based problems as described with Υ. Θ reflects the classifier type (LCN, LCPN, LCL, or GC), as described above.

The here described framework was initially introduced to categorize any hierarchical classification problem. However, for the goal of reducing the energy footprint of a model, the possible model structure usually boils down to

$$(\Delta,\Xi,\Omega,\Theta) = (MPP, MLNP, T, LCPN) \ . \tag{9.5}$$

In this setting, problems like the above-mentioned condition monitoring use case can use hierarchical machine learning to save energy. In the example, the no-fault label could be reached by either the first or second classifier, making the model MPP. Additionally, MLNP is mandatory for saving energy, as we want to be able to stop calculating at any helpful point in the hierarchy. Even though a tree model is not entirely mandatory, allowing additional links to nodes other than child nodes often makes no sense and just improves complexity. Additionally, we want to be able to tailor each classifier in the hierarchy to a specific task. Allowing for additional links would bypass this principle. An LCPN classifier is the most flexible approach for structures, permitting asymmetric taxonomies with branching while allowing multi-class classifiers at each node.

With hierarchical predictions, another problem arises with partially correct predictions, which normally would result in a low score with standard metrics. In the condition monitoring example, the model might have correctly classified the fault but got the severity wrong. To approach this question, the authors recommend using the hierarchical precision, recall, and f-measure

introduced by [80], where $\widehat{P}_i$ denotes a set consisting of the most specific classes predicted for the sample i as well as all its ancestors, and $\widehat{T}_i$ the set holding the true most specific classes for i. Considering the ancestors of class labels, these measures address the problem of partially correct classifications in the final scores.

$$hP = \frac{\sum_i |\widehat{P}_i \cap \widehat{T}_i|}{\sum_i |\widehat{P}_i|}, \; hR = \frac{\sum_i |\widehat{P}_i \cap \widehat{T}_i|}{\sum_i |\widehat{T}_i|}, \; hF = \frac{2 \cdot hP \cdot hR}{hP + hR} \; . \tag{9.6}$$

The usability of hierarchical machine learning for energy saving, specifically for small sensor systems, has also been identified by [81] Here, the authors introduce a general description of a cascaded LCL model tailored towards lazily triggered stages. They differentiate between simple wake-up mechanisms, a two-stage hierarchy with only anomaly detection, and more sophisticated cascades. For the latter, they introduce a pass-on label, which a classifier can use to defer a decision. In this case, the decision is passed on to the next level, triggering further computations. A model's pass-on rate (POR) directly influences the hierarchical metrics described above. A POR of 1 would mean the decision is passed on to the final classifier for each sample. This leads to the hierarchical precision and recall metrics equaling their non-hierarchical counterparts for the last classifier.Additionally, the authors assign each stage a cost measure, as more memory is needed with increasing stages. Combining these thoughts about POR, Cost, and hierarchical metrics, the authors introduce an upper bound for optimizing a hierarchy wrt. computational costs and memory. With this upper bound and cost measure, they carry out proof of concept with synthetic test cases to optimize the number of stages in different scenarios. These scenarios are compiled to include class distributions ranging from equal to extremely skewed, where some classes are more likely than others. The authors conclude that optimisation can find lower-cost hierarchical models for more skewed class distribution. This effect becomes apparent when considering the locality of the most likely prediction point in the hierarchy. As some labels are more likely than others, detecting these in the early stages can save a lot of energy. For the condition monitoring example, this effect can be observed for the first two classifiers. If the no-fault label occurs most often, the hierarchy uses only the first (or, in some cases, the second) classifier. This leads to enormous energy savings as the computational complexity of each model should rise with the levels in the cascade. This phenomenon clearly shows the inherent data dependency of a hierarchical model regarding the reduction in energy consumption. The more

uniform the data distribution of the underlying problem is, the less effective a hierarchical approach becomes.

One can also consider mixing different classifier types in the hierarchy to further elevate the benefits of hierarchical processing, as some decisions may need more complex models than others. However, with that idea in mind, the question of how to pick each model in the hierarchy arises. The authors in [82], [83] approach this optimization question with reinforcement learning. They train an agent to pick a model from a set of available models for each node in the hierarchy of an LCPN model. As the reward function, they use a cost measure based on the computational complexity of each possible model type, combined with the accuracy of the complete hierarchy. Therefore, the agent learns to make a trade-off between accuracy and energy consumption, which, in essence, leads to a multi-objective optimization problem. To improve that result, the computational complexity is replaced with a hardware-in-the-loop approach to accurately measure the resulting energy consumption during the agent's training in [84]. While this work mostly validates the findings of [82], it shows slightly different behavior of the found agents. With real measurements, the agent picks an MLP more often, while the agent with the complexity approximation tends to use Random Forests. This behavior might be linked to compiler optimizations and caching, which can be beneficial for some models. This result shows that energy measurements should always be included to some degree when optimizing for a constrained device, as a theoretical measure cannot cover every possible factor of real-life systems.

9.6.2 Distributed Computing

Another area where a hierarchical model can be beneficial is distributed computing for IoT devices. [85] introduces a distributed hierarchical inference approach, where some nodes in the hierarchy are computed locally and others in the cloud. This has the benefit of saving communication costs that might be unnecessary. They look at a scenario with multiple connected IoT appliances, which all have some parts of the information needed for home automation. In a classic approach, all these appliances would send data to a central hub or cloud infrastructure at every time step, which leads to an immense communication overhead. However, some decisions are possible without knowing the context of the other devices. In the hierarchical context, these decisions can be made with small models running locally on the IoT device. If this is not possible, the devices may pass the prediction to the

cloud, triggering the next node in the hierarchy. The authors test this approach with three data sets from three application domains (urban energy demand, human activity, and server performance) with a decrease in system energy consumption of 62% for a taxonomy of MLPs.It should be noted that the authors used a DAG instead of a tree architecture for their models, as some nodes in the hierarchy connect to the same child node. This is because the cloud node(s) need information from multiple parent nodes because of the nature of IoT applications, where usually numerous sensors are required for a decision. Therefore, a typical hierarchy becomes a flipped tree and can only be described with a DAG.

The described benefits of IoT devices are tested in a case study in [86], where the authors use a physical system to measure the energy consumption of an LCL model for human activity recognition. The model uses multiple different classifiers in a cascade that are partially executed on the device. As a novelty, they introduce an offload controller that decides if the model should be run on the device or offloaded to a central hub. This controller consists of an additional classifier that has been trained on binary labels, which are extracted from the original multi-class problem. In addition, the controller uses only the features necessary to compute the next hierarchical layer that might be offloaded. This means that the controller performs a reduced complexity pre-evaluation of the problem solved by the following levels to decide if the computation needs to be offloaded. The paper shows energy reduction of 3 times over the baseline of a purely offloaded system, including communication costs, while improving accuracy slightly.

Taking the idea of split computing further, [87] introduces a dynamic split mechanism to split the computation between different models and a single DNN across multiple devices. In their proposed scenario, various IoT devices are connected to a hub. Based on the currently available network bandwidth, the introduced system can decide to hand off the computation to the hub. In addition, the authors train an agent with Q-learning to dynamically assign the network type (Wi-Fi, cellular, Bluetooth, etc.) for the data transfer of the remaining network layers. The authors can decrease the inference time by 13.5% compared to pure on-device execution, including communication time. However, it should be noted that no energy metrics are compared here, so while more time efficient, the shown approach might still draw more energy than pure on-device execution. Nonetheless, the approach should be considered when designing a hierarchical model.

9.6.3 Early-Exit Neural Networks

Most hierarchies mentioned before were constructed of classical machine-learning approaches like Support Vector Machines or Random Forests. Even though some architectures also used MLPs in their taxonomies, the work in [88] converts the concept of lazily triggered computations to Convolutional Neural Networks. Like the NMLP principle for classic hierarchies, the BranchyNet introduced here includes exit points in the neural network architecture (cf. Figure 9.6). The network can stop computing if a decision can be made in some earlier parts of the network based on a confidence score. Like nodes in the classical hierarchy, the authors define a branch as a non-overlapping subset of the complete network, with one entry point and multiple exit points. To train BranchyNet, the authors use a weighted formulation of the loss function which sums over the loss functions of each exit n with a weighting factor w_n. In this formulation, any arbitrary loss function that uses a one-hot encoded output vector for the prediction and ground truth $\hat{y}$ or y, respectively. θ describes the network's parameter of the branch before the exit point n. The choice of w_n influences the exit strategy of the network. Giving a high value for earlier exits causes more discriminative feature learning in the earlier branches. This leads to more early exiting but might compromise the accuracy. Contrarily, the network's accuracy increases with higher weights for the later exits, but exiting in the earlier stages becomes rare. During training, the network is trained as one unit without exiting early. However, for the inference, the authors use the entropy of the output at each exit point to determine if the calculation can be stopped. The computation is stopped if it is lower than a threshold T_n, reducing the inference time. Again, T_n is a hyperparameter that influences the trade-off between accuracy and inference time but might also be set automatically based on experimental results. The authors use three basic CNN architectures (LeNet, AlexNet, and ResNet) with added exit points to test the approach. They sweep through various values for T, which allows for a trade-off analysis of accuracy vs. runtime. For all three basic architectures, the BranchyNet approach leads to an optimal vector T that achieves a 2x-6x average runtime decrease while maintaining similar or superior accuracy compared to the non-branched networks. In addition to these results, the authors also observe a decrease in cache misses with more aggressive values for T. This leads to the conclusion that with a smart branch design, this behavior could be exploited to use the cache of a system effectively.

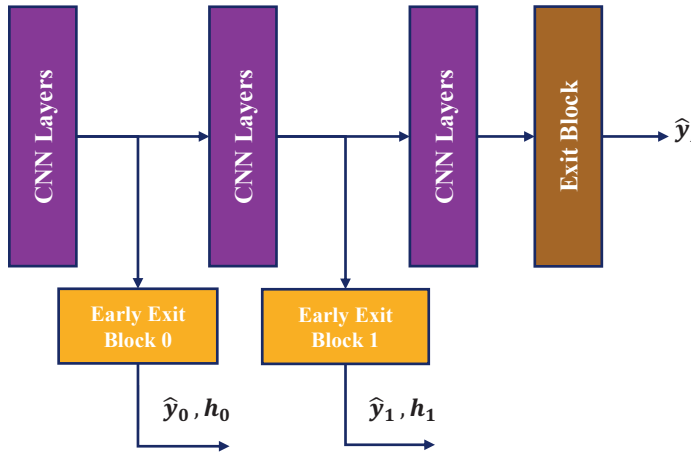

Figure 9.6 General structure of an Early-Exit CNN. After some CNN layers, exits can stop the computation based on a confidence metric

$$L_{branchynet}\left(\hat{y}, y, \theta\right) = \sum_{n=1}^{N} w_n L\left(\widehat{y_{exit_n}}, y; \theta\right) \ . \tag{9.7}$$

A similar approach is taken in [89]. The authors also introduce exit points in the network, but instead of the entropy, they use a learnable decision function $\gamma_n\left(\sigma_n\left(x\right)\right)$ as an exit strategy. Depending on the output of the previous CNN-layer σ_n, the decision function can decide to exit the execution. In addition, similar to mixed hierarchies with different model architectures, the authors propose a selection algorithm to fill branches in the early exit network with different architectures. They argue that easier classes might be classifiable with a simple AlexNet while others might need something more advanced like ResNet. With this approach, the authors achieve a speed-up of 1.5x to 7.53x, depending on the acceptable accuracy degradation. For example, the adaptive strategy can have a speed-up of around 2x compared to ResNet50 while staying in the range of 1% in terms of accuracy loss.

An additional implementation of an early exit strategy can be found in [90] called MSDNet. In their approach, branches are interconnected with a small classification network. The authors identify two problems with intermediate exits in standard architecture. Without any modifications, the early layers lack coarse-level features, which leads to worse classification results for earlier exists. Therefore, they introduce additional coarse features by adding parallel convolutional layers that calculate features on coarser scales. These features are concatenated with the ones from finer scales for

intermediate classification. This addition leads to a substantial increase in accuracy for the exists.

The second problem is the interference of earlier classifiers with later ones. The authors noticed that adding intermediate classifiers for early exiting harms the accuracy at the final exit. Therefore, like a densely connected network, they add connections between earlier and later convolutional layers. In this way, the network should be able to forward information from earlier stages, which would otherwise be lost. After this change, the final exit becomes independent from the intermediate exits.

To optimize Early-Exit-NNs for on-device execution, [91] introduces an on-device transfer learning approach to fine-tune the intermediate exits in the field. The authors first train a generic MSDNet with equidistantly placed intermediate exits. On-device, they personalize the global pre-trained model by training only the intermediate exits. Due to the structure of MSDNet with bypass connection spanning the whole network, re-training the intermediate exits does not affect the final exit's performance. Therefore, they can use the classification result of the final prediction to teach the intermediate exists even if no labeled data is available in the field. They continue the personalization by tuning the threshold to decide if the calculation can be stopped at an intermediate exit. Like Multi-Objective Optimization, the authors use a small user-input data set to obtain a Pareto front to fine-tune the threshold. With this approach, the authors could measure a significant improvement in inference time and accuracy.

In addition, further field testing is done in [92] with a modified version of BranchyNet. Instead of solely relying on the entropy to decide on the next branch, the authors include energy-aware criteria based on the battery level. Only a shallow branch is computed if the remaining capacity is lower than a threshold. Otherwise, the decision is based on the entropy, as described in the BranchyNet paper. The authors also conduct thorough energy profiling and hardware-aware optimization steps that are out of the scope of this work. Nonetheless, it shows the importance of optimising the target platform, as the authors can deploy their approach to a processor with just 24kB SRAM and 128kB flash memory. In addition, they are also able to predict the battery size needed for their system.

9.7 Discussion

The highlighted literature shows that compared to flat classifiers, cascaded systems can decrease the latency by a substantial amount, which often

translates into a decrease in energy consumption. The type of hierarchy can vary from simple cascaded structures to DAG or tree structures with multiple branches. The key here is always the uneven distribution of data in real-world systems, where some labels are either more frequent or easier to predict than others. This leads to a system that can abort computation early if certain criteria are met, which can be boiled down to the NMLNP property of hierarchical models. When interpreting nodes in a hierarchical model as sub-sets of a Neural Network, the NMLNP property can also be seen in early-exit-NNs, which also have the property to abort computation with intermediate exits. These architectures rely on course features computed by early CNN layers, which, for some labels, are enough to come to a correct prediction. Based on a confidence score, the system can stop computing at intermediate exits, which boils down to a hyperparameter optimization problem on thresholds. Depending on the thresholds, the network can trade between accuracy (regular usage of later exits) and energy efficiency (frequent usage of earlier exits).

Acknowledgements

This work was partially supported by the projects Center for Analytics-Data-Applications (ADA-Center) and MANOLO. ADA-Center is funded by the Bavarian Ministry of Economic Affairs, Regional Development and Energy within the framework of "BAYERN DIGITAL II" (20-3410-2-9-8). MANOLO is funded by the European Commission under the Horizon Europe programme Grant Agreement No.101135782.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," Conference on Computer Vision and Pattern Recognition, 2016.

[2] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," Journal of Machine Learning Research - Proceedings Track, vol. 9, pp. 249–256, 2010.

[3] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," Conference on Computer Vision and Pattern Recognition, 2017. 1003

[4] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," Conference on Computer Vision and Pattern Recognition, 2018.

[5] A. G. Howard, M. Sandler, L.-C. C. Grace Ch and, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam, "Searching for mobilenetv3," Conference on Computer Vision and Pattern Recognition, 2019.

[6] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in Conference on Computer Vision and Pattern Recognition, 2018.

[7] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in International Conference on Machine Learning, 2019.

[8] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in Conference on Computer Vision and Pattern Recognition, 2019.

[9] B. Yang, G. Bender, Q. V. Le, and J. Ngiam, "Condconv: Conditionally parameterized convolutions for efficient inference," in International Conference on Neural Information Processing Systems, 2019.

[10] J. Dai, H. Qi, Y. Xiong, Y. Li, G. Zhang, H. Hu, and Y. Wei, "Deformable convolutional networks," in International Conference on Computer Vision (ICCV), 2017.

[11] X. Zhu, H. Hu, S. Lin, and J. Dai, "Deformable convnets v2: More deformable, better results," in 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2019.

[12] J. Schmidhuber, "Learning to control fast-weight memories: An alternative to dynamic recurrent networks," Neural Computation, 1992.

[13] D. Ha, A. M. Dai, and Q. V. Le, "Hypernetworks," in International Conference on Learning Representations, 2017.

[14] B. De Brabandere, X. Jia, T. Tuytelaars, and L. Van Gool, "Dynamic filter networks," in International Conference on Neural Information Processing Systems, 2016

[15] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in International Conference on Machine Learning, 2013.

[16] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-all: Train one network and specialize it for efficient deployment," in International Conference on Learning Representations, 2020.

[17] B. Zoph and Q. Le, "Neural architecture search with reinforcement learning," in International Conference on Learning Representations, 2017.

[18] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," in International Conference on Learning Representations, 2017.

[19] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in Proceedings of the AAAI conference on artificial intelligence, 2019.

[20] C. White, W. Neiswanger, and Y. Savani, "Bananas: Bayesian optimization with neural architectures for neural architecture search," in Proceedings of the AAAI Conference on Artificial Intelligence, 2021.

[21] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," in International Conference on Learning Representations, 2019.

[22] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in IEEE Conference on Computer Vision and Pattern Recognition, CVPR, 2019.

[23] X. Shen, Y. Wang, M. Lin, Y. Huang, H. Tang, X. Sun, and Y. Wang, "Deepmad: Mathematical architecture design for deep convolutional neural network," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2023.

[24] A. Zela, T. Elsken, T. Saikia, Y. Marrakchi, T. Brox, and F. Hutter, "Understanding and robustifying differentiable architecture search," in International Conference on Learning Representations, 2020.

[25] H. Wang, C. Ge, H. Chen, and X. Sun, "Prenas: Preferred one-shot learning towards efficient neural architecture search," arXiv preprint arXiv:2304.14636, 2023.

[26] J. Mellor, J. Turner, A. Storkey, and E. J. Crowley, "Neural architecture search without training," in International Conference on Machine Learning. PMLR, 2021, pp. 7588–7598.

[27] M. Lin, P. Wang, Z. Sun, H. Chen, X. Sun, Q. Qian, H. Li, and R. Jin, "Zen-nas: A zero-shot nas for high-performance image recognition," in Proceedings of the IEEE/CVF International Conference on Computer Vision, 2021, pp. 347–356.

[28] M. S. Abdelfattah, A. Mehrotra, Ł. Dudziak, and N. D. Lane, "Zero-cost proxies for lightweight nas," International Conference on Learning Representations (ICLR), 2021.

[29] N. Lee, T. Ajanthan, and P. H. Torr, "Snip: Single-shot network pruning based on connection sensitivity," International Conference on Learning Representations (ICLR), 2019.
[30] C. Wang, G. Zhang, and R. Grosse, "Picking winning tickets before training by preserving gradient flow," International Conference on Learning Representations (ICLR), 2020.
[31] C. White, M. Khodak, R. Tu, S. Shah, S. Bubeck, and D. Dey, "A deeper look at zero-cost proxies for lightweight nas," ICLR Blog Track, 2022.
[32] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-II," IEEE Transactions on Evolutionary Computation, 2002.
[33] Z. Wang, T. Luo, M. Li, J. T. Zhou, R. S. M. Goh, and L. Zhen, "Evolutionary multi-objective model compression for deep neural networks," IEEE Computational Intelligence Magazine, 2021.
[34] J. Knowles, "Parego: a hybrid algorithm with on-line landscape approximation for expensive multiobjective optimization problems," IEEE Transactions on Evolutionary Computation, 2006.
[35] R. Bellman, "Dynamic programming," Science, vol. 153, no. 3731, pp. 34–37, 1966.
[36] C. Liu, L.-C. Chen, F. Schroff, H. Adam, W. Hua, A. Yuille, and L. Fei-Fei, "Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation," in IEEE Conference on Computer Vision and Pattern Recognition, CVPR, 2019.
[37] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. de Freitas, "Predicting parameters in deep learning," Advances in Neural Information Processing Systems, 2013.
[38] S. Hanson and L. Pratt, "Comparing biases for minimal network construction with back-propagation," Advances in Neural Information Processing Systems, vol. 1, 1988.
[39] Y. LeCun, J. Denker, and S. Solla, "Optimal brain damage," Advances in neural information processing systems, vol. 2, 1989.
[40] B. Hassibi and D. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," Advances in neural information processing systems, vol. 5, 1992.
[41] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," in International Conference on Learning Representations, 2016.

[42] M. Lin, R. Ji, Y. Zhang, B. Zhang, Y. Wu, and Y. Tian, "Channel pruning via automatic structure search," in Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence, 2021, pp. 673–679.

[43] X. Xu, M. S. Park, and C. Brick, "Hybrid pruning: Thinner sparse networks for fast inference on edge devices," arXiv preprint arXiv:1811.00482, 2018.

[44] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," in International Conference on Learning Representations, 2019.

[45] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," Advances in neural information processing systems, vol. 28, 2015.

[46] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," arXiv preprint arXiv:1510.00149, 2015.

[47] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," arXiv preprint arXiv:1607.03250, 2016.

[48] S.-K. Yeom, P. Seegerer, S. Lapuschkin, A. Binder, S. Wiedemann, K.-R. Müller, and W. Samek, "Pruning by explaining: A novel criterion for deep neural network pruning," Pattern Recognition, vol. 115, p. 107899, 2021.

[49] R. Xu, S. Luan, Z. Gu, Q. Zhao, and G. Chen, "LRP-based policy pruning and distillation of reinforcement learning agents for embedded systems," in IEEE 25th International Symposium On Real-Time Distributed Computing (ISORC), 2022, pp. 1–8.

[50] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," arXiv preprint arXiv:1710.01878, 2017.

[51] M. Kohlbrenner, A. Bauer, S. Nakajima, A. Binder, W. Samek, and S. Lapuschkin, "Towards best practice in explaining neural network decisions with lrp," in International Joint Conference on Neural Networks. IEEE, 2020, pp. 1–7.

[52] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," arXiv preprint arXiv:1806.08342, 2018. S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in International conference on machine learning. PMLR, 2015, pp. 1737–1746.

[53] M. Nagel, R. A. Amjad, M. Van Baalen, C. Louizos, and T. Blankevoort, "Up or down? adaptive rounding for post-training quantization," in International Conference on Machine Learning. PMLR, 2020, pp. 7197–7206.

[54] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC). IEEE, 2014, pp. 10–14.

[55] J. Johnson, "Rethinking floating point for deep learning," arXiv preprint arXiv:1811.01721, 2018.

[56] D. L. N. Hettiarachchi, V. S. P. Davuluru, and E. J. Balster, "Integer vs. floating-point processing on modern fpga technology," in 2020 10th Annual Computing and Communication Workshop and Conference (CCWC). IEEE, 2020, pp. 0606–0612.

[57] D. Zhang, J. Yang, D. Ye, and G. Hua, "Lq-nets: Learned quantization for highly accurate and compact deep neural networks," in Proceedings of the European conference on computer vision (ECCV), 2018, pp. 365–382.

[58] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," arXiv preprint arXiv:2210.17323, 2022.

[59] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2018, pp. 2704–2713.

[60] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," arXiv preprint arXiv:1802.05668, 2018.

[61] J. H. Lee, S. Ha, S. Choi, W.-J. Lee, and S. Lee, "Quantization for rapid deployment of deep neural networks," arXiv preprint arXiv:1810.05488, 2018.

[62] R. Banner, Y. Nahshan, and D. Soudry, "Post training 4-bit quantization of convolutional networks for rapid-deployment," Advances in Neural Information Processing Systems, vol. 32, 2019.

[63] M. G. d. Nascimento, R. Fawcett, and V. A. Prisacariu, "Dsconv: efficient convolution operator," in Proceedings of the IEEE/CVF International Conference on Computer Vision, 2019, pp. 5148–5157.

[64] B. Darvish Rouhani, D. Lo, R. Zhao, M. Liu, J. Fowers, K. Ovtcharov, A. Vinogradsky, S. Massengill, L. Yang, R. Bittner et al., "Pushing the limits of narrow precision inferencing at cloud scale with microsoft

floating point," Advances in neural information processing systems, vol. 33, pp. 10 271–10 281, 2020.
[65] G. Xiao, J. Lin, M. Seznec, J. Demouth, and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," arXiv preprint arXiv:2211.10438, 2022.
[66] Y. Choukroun, E. Kravchik, F. Yang, and P. Kisilev, "Low-bit quantization of neural networks for efficient inference," in 2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW). IEEE, 2019, pp. 3009–3018.
[67] P. Wang, Q. Chen, X. He, and J. Cheng, "Towards accurate post-training network quantization via bit-split and stitching," in International Conference on Machine Learning. PMLR, 2020, pp. 9847–9856.
[68] Z. Yao, R. Yazdani Aminabadi, M. Zhang, X. Wu, C. Li, and Y. He, "Zeroquant: Efficient and affordable post-training quantization for large- scale transformers," Advances in Neural Information Processing Sys- tems, vol. 35, pp. 27 168–27 183, 2022.
[69] Y. Bondarenko, M. Nagel, and T. Blankevoort, "Understanding and overcoming the challenges of efficient transformer quantization," arXiv preprint arXiv:2109.12948, 2021.
[70] Y. Cai, Z. Yao, Z. Dong, A. Gholami, M. W. Mahoney, and K. Keutzer, "Zeroq: A novel zero shot quantization framework," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 13 169–13 178.
[71] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "Pact: Parameterized clipping activation for quantized neural networks," arXiv preprint arXiv:1805.06085, 2018. E. Park, S. Yoo, and P. Vajda, "Value-aware quantization for training and inference of neural networks," in Proceedings of the European Conference on Computer Vision (ECCV), 2018, pp. 580–595.
[72] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. Van Baalen, and T. Blankevoort, "A white paper on neural network quantization," arXiv preprint arXiv:2106.08295, 2021.
[73] Y. Bengio, N. Lt'eonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," arXiv preprint arXiv:1308.3432, 2013.
[74] P. Yin, J. Lyu, S. Zhang, S. Osher, Y. Qi, and J. Xin, "Understanding straight-through estimator in training activation quantized neural nets," arXiv preprint arXiv:1903.05662, 2019.

[75] I. Hubara, Y. Nahshan, Y. Hanani, R. Banner, and D. Soudry, "Improving post training neural quantization: Layer-wise calibration and integer programming," arXiv preprint arXiv:2006.10518, 2020.

[76] M. Nagel, M. v. Baalen, T. Blankevoort, and M. Welling, "Data-free quantization through weight equalization and bias correction," in Proceedings of the IEEE/CVF International Conference on Computer Vision, 2019, pp. 1325–1334.

[77] A. Finkelstein, U. Almog, and M. Grobman, "Fighting quantization bias with bias," arXiv preprint arXiv:1906.03193, 2019.

[78] E. Meller, A. Finkelstein, U. Almog, and M. Grobman, "Same, same but different: Recovering neural network quantization error through weight factorization," in International Conference on Machine Learning. PMLR, 2019, pp. 4486–4495.

[79] C. N. Silla and A. A. Freitas, "A survey of hierarchical classification across different application domains," Data Min Knowl Disc, vol. 22, pp. 31–72, 2011.

[80] S. Kiritchenko and F. Famili, "Functional annotation of genes using hierarchical text categorization," Proceedings of BioLink SIG, ISMB, 01 2005.

[81] K. Goetschalckx, B. Moons, S. Lauwereins, M. Andraud, and M. Verhelst, "Optimized hierarchical cascaded processing; optimized hierarchical cascaded processing," IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 8, 2018. [Online]. Available: http://www.ieee.org/publicationsstandards/publications/rights/index.html

[82] S. Adams, R. Meekins, P. A. Beling, K. Farinholt, N. Brown, S. Polter, and Q. Dong, "Hierarchical fault classification for resource constrained systems," Mechanical Systems and Signal Processing, vol. 134, p. 106266, Dec. 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0888327019304819

[83] S. Adams, T. Cody, P. A. Beling, S. Polter, and K. Farinholt, "Hierarchical classification for unknown faults; hierarchical classification for unknown faults," 2020 IEEE International Conference on Prognostics and Health Management (ICPHM), 2020.

[84] J. Wissing, S. Scheele, A. Mohammed, D. Kolossa, and U. Schmid, "Himledge – energy-aware optimization for hierarchical machine learning," in Advanced Research in Technologies, Information, Innovation and Sustainability, T. Guarda, F. Portela, and M. F. Augusto, Eds. Cham: Springer Nature Switzerland, 2022, pp. 15–29.

[85] A. Thomas, Y. Guo, Y. Kim, B. Aksanli, A. Kumar, T. S. Rosing, and U. S. Diego, "Hierarchical and distributed machine learning inference beyond the edge; hierarchical and distributed machine learning inference beyond the edge," 2019 IEEE 16th International Conference on Networking, Sensing and Control (ICNSC), 2019.

[86] F. Samie, L. Bauer, and J. Henkel, "Hierarchical classification for constrained iot devices: A case study on human activity recognition," IEEE Internet of Things Journal, vol. 7, pp. 8287–8295, 9 2020,

[87] J. Karjee, K. Anand, V. N. Bhargav, P. S. Naik, R. B. V. Dabbiru, and N. Srinidhi, "Split computing: Dynamic partitioning and reliable communications in iot-edge for 6g vision." IEEE, 8 2021, pp. 233–240. [Online]. Available: https://ieeexplore.ieee.org/document/9590311/

[88] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," vol. 0. Institute of Electrical and Electronics Engineers Inc., 1 2016, pp. 2464–2469.

[89] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama, "Adaptive neural networks for efficient inference," in Proceedings of the 34th International Conference on Machine Learning - Volume 70, ser. ICML'17. JMLR.org, 2017, p. 527–536.

[90] G. Huang, D. Chen, T. Li, F. Wu, L. van der Maaten, and K. Q. Weinberger, "Multi-scale dense networks for resource efficient image classification," in International Conference on Learning Representations, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:3475998

[91] I. Leontiadis, S. Laskaridis, S. I. Venieris, and N. D. Lane, "It's always personal: Using early exits for efficient on-device cnn personalisation." Association for Computing Machinery, Inc, 2 2021, pp. 15–21.

[92] Y. Li, Y. Wu, X. Zhang, J. Hu, and I. Lee, "Energy-aware adaptive multi-exit neural network inference implementation for a millimeter-scale sensing system," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 30, pp. 849–859, 7 2022.