CONTRIBUTED PAPER

Efficient Acceleration of Deep Learning Inference on Resource-Constrained Edge Devices: A Review

This article provides a comprehensive review of the state-of-the-art tools and techniques for efficient edge inference, a vital element of artificial intelligence on edge.

By Md. Maruf Hossain Shuvo, *Graduate Student Member IEEE*, Syed Kamrul Islam, *Senior Member IEEE*, Jianlin Cheng, *Member IEEE*, and Bashir I. Morshed, *Senior Member IEEE*

ABSTRACT | Successful integration of deep neural networks (DNNs) or deep learning (DL) has resulted in breakthroughs in many areas. However, deploying these highly accurate models for data-driven, learned, automatic, and practical machine learning (ML) solutions to end-user applications remains challenging. DL algorithms are often computationally expensive, power-hungry, and require large memory to process complex and iterative operations of millions of parameters. Hence, training and inference of DL models are typically performed on high-performance computing (HPC) clusters in the cloud. Data transmission to the cloud results in high latency, round-trip delay, security and privacy concerns, and the inability of real-time decisions. Thus, processing on edge devices can significantly reduce cloud transmission cost. Edge devices are end devices closest to the user, such as mobile phones, cyber–physical systems (CPSs), wearables, the Internet of Things (IoT), embedded and autonomous systems, and intelligent sensors. These devices have limited memory, computing resources, and power-handling capability. Therefore, optimization techniques at both the hardware and software levels have been developed to handle the DL deployment efficiently on the edge. Understanding the existing research, challenges, and opportunities is fundamental to leveraging the next generation of edge devices with artificial intelligence (AI) capability. Mainly, four research directions have been pursued for efficient DL inference on edge devices: 1) novel DL architecture and algorithm design; 2) optimization of existing DL methods; 3) development of algorithm–hardware codesign; and 4) efficient accelerator design for DL deployment. This article focuses on surveying each of the four research directions, providing a comprehensive review of the state-of-the-art tools and techniques for efficient edge inference.

KEYWORDS | Algorithm–hardware codesign; artificial intelligence (AI); artificial intelligence on edge (edge-AI); deep learning (DL); model compression; neural accelerator.

Manuscript received 22 February 2022; revised 14 September 2022 and 17 November 2022; accepted 30 November 2022. Date of publication 14 December 2022; date of current version 12 January 2023. This work was supported in part by the College of Engineering, University of Missouri, Columbia, MO, USA. *(Corresponding author: Md. Maruf Hossain Shuvo.)*

Md. Maruf Hossain Shuvo and **Syed Kamrul Islam** are with the Analog/Mixed Signal VLSI and Devices Laboratory (AVDL), Department of Electrical Engineering and Computer Science, University of Missouri, Columbia, MO 65211 USA (e-mail: msp8c@umsystem.edu; islams@missouri.edu).

Jianlin Cheng is with the Bioinformatics and Machine Learning Laboratory (BML), Department of Electrical Engineering and Computer Science, University of Missouri, Columbia, MO 65211 USA (e-mail: chengji@missouri.edu).

Bashir I. Morshed is with the Cyber Physical Systems (CPS) Laboratory, Department of Computer Science, Texas Tech University, Lubbock, TX 79409 USA (e-mail: bmorshed@ttu.edu).

Digital Object Identifier 10.1109/JPROC.2022.3226481

I. Introduction

Machine learning (ML) algorithms extract and learn characteristic features from data for automatic decision-making. Deep learning (DL) is one of the subfields of ML that has enabled many advanced technologies. DL algorithms automatically learn from simple to complex features while moving from shallow to deeper layers [1].

A neural network (NN) architecture also termed a multilayer perceptron (MLP) is presented in Fig. 1. The three main kinds of layers of an NN are the input layer,

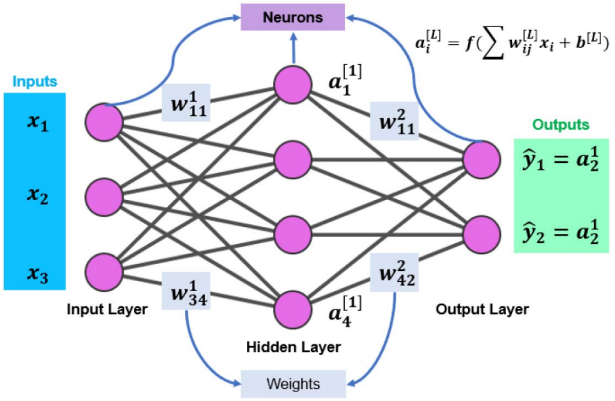


Fig. 1. *Overview of the terminology, operations, and processing in a basic NN model. Notations: x = example of training data, w_{ij}^L = weights connecting neuron i to j in layer L, $a_i^{[L]}$ = activation of neuron i in layer L, f = activation function, $b^{[L]}$ = bias in layer L, and $\hat{y}$ = predicted output.*

hidden layers, and output layer. Input signals propagate through the neurons in the hidden layers accumulating learned information. Weights control the strength of the connection between two neurons deciding the influence of input on the output. The weighted sums calculated in each layer are known as the activations. If an NN consists of more than three hidden layers, the model is termed a deep NN (DNN) or DL [2]. The human-level accuracy may result from the ability to add many hidden layers.

The basic training and inference process of a DL model is illustrated in Fig. 2. Training is the refinement of the output through an iterative learning process that includes both forward and backward propagation. Weights and biases are the parameters learned and updated through the training process. Computing the predictions from unseen data using the learned parameters is called inference or prediction. Each inference task consists of only the forward propagation. Training a DL model is computationally more expensive than inference. During the training phase, the parameters are updated through optimization processes to minimize a loss function. An efficient way to update the weights and biases is backpropagation, which passes outputs backward through the network to determine the effects of each parameter on the loss to refine the parameters.

The deployment of DL models for inference can be of two main types: inference on cloud and inference on edge. Cloud-based processing requires data transmission from the acquisition site to the cloud. On the other hand, processing on edge performs inference near the source of data. Three popular use cases of DL training and inference are: 1) training and inference on the cloud; 2) training and inference on edge; and 3) training on the cloud and inference on edge. Training and inference on the cloud focus on highly accurate and deeper models with a large dataset. Edge devices have many resource constraints, such as limited memory, computation, and power budget. Thus, training on edge is of limited capacity. The focus of this article is the inference on edge cases. For inference on edge, the models are pretrained to be deployed for inference on artificial intelligence on edge (edge-AI) devices. There are numerous use cases in security, surveillance, healthcare, autonomous driving, and so on, where real-time inference on edge is necessary. For instance, the intensive care unit (ICU) requires real-time signal processing and decision-making. A closed-loop system can remove human intervention and error using AI to balance vital physiological signals within a specific range. Smart wearable devices use ambient intelligence to process several sensor data on edge to improve assisted living by quickly detecting anomalies (e.g., a fall or a fire) and taking immediate actions. Business and commercial values of edge-AI are apparent in the manufacturing and industrial sectors. Embedded sensors of machines in an industry can apply DL for predictive analysis of equipment failure. This automation can increase productivity by removing off-line inspection of machinery at regular intervals.

Hardware acceleration for efficient and accurate DL inference on edge has many advantages over cloud-based computing. Fast and real-time use cases are possible by reducing the latency as data are processed near the source. Better security and bandwidth efficiency are ensured due to data not being transmitted to the cloud. Other benefits of edge inference (EI) are scalability and reliability. In EI, the data collection, storage, and processing are distributed across different parts, making it difficult for any single disruption (e.g., cybersecurity threats and power outages) to take down the system. As sensitive data can be processed locally and in real-time without streaming it to the cloud, privacy is ensured. Therefore, many applications of AI are migrating from the cloud to embedded edge devices. Virtual assistants, such as Alexa, Siri, and Google Assistant, use on-chip speech recognition as part of the processing for real-time assistance. Smartphones use embedded AI to create a better picture using computational imaging techniques taking input from multiple lenses. In smart televisions, AI upscales the high-definition content to recreate the missing details. Wearable devices with integrated AI are now facilitating the monitoring and processing of vital signs and fitness information to track or detect various diseases. In a hospital environment, edge-AI performs inventory management, remote monitoring of patients, thermal screening, and disease prediction. Unmanned aerial vehicles (UAVs) can ensure safety inspections in remote and harsh environments (traffic, construction, fire, cartography, security, and so on) with on-device processing facilitated by AI. Robots with AI provide efficient manufacturing having high precision and scalability in industrial applications. Besides, manufacturing flaws can be detected using AI-enabled cameras for quality control that can otherwise be impossible in human eyes. Fingerprint detection, face identification security, fraud detection, and autonomous driving are a few practical applications brought by edge-AI.

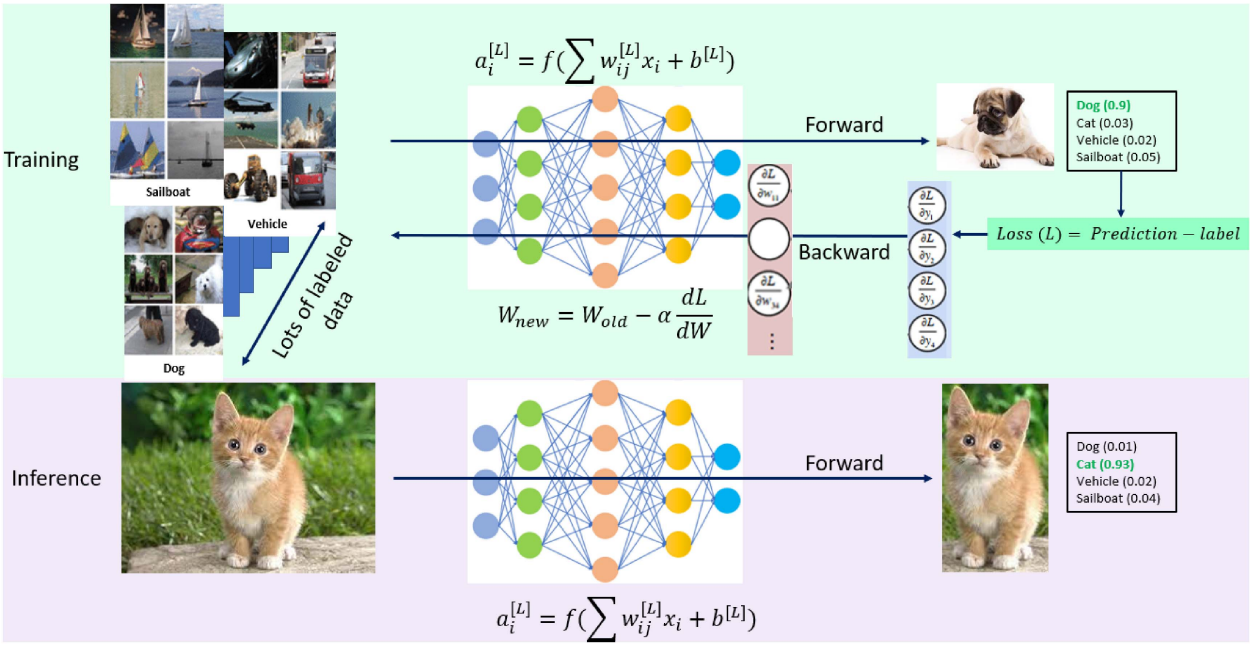


Fig. 2. *Overview of the DL training and inference process. Training is an iterative process of both forward and backward propagation. The inference is a single forward propagation to find the output for a given input using the parameters learned during training. Notations:* α = *learning rate,* W_{new} = *updated weight matrix,* W_{old} = *current weight matrix, and* (dL/dW) = *gradient of loss L.*

In addition, edge-AI with video analytics facilitates automated product replacement, arrangement in-store, and placement and layout in the home environment before making a purchase. The edge-AI cameras can determine reckless drivers, emergency conditions, and road blockage to ensure safety and efficiency. Self-driving cars are another practical example of edge-AI, which can prevent fatal accidents through on-site processing of data from various sensors for detecting pedestrians, roadblocks, and recognition of traffic signals and surrounding vehicles. This trend brings new design challenges. Some of the challenges are programmability, real-time performance requirements, and the power consumption of the hardware to adapt to the changes and evolution of the new DL algorithms. Besides, many applications on the edge require real-time outputs within a specific latency limit. Energy efficiency is another crucial requirement in the design of EI. Customized and dedicated application-specific design improves energy efficiency but is limited in flexibility and programmability. Therefore, edge devices contemplate the best tradeoffs between performance, power, energy efficiency, latency, cost, and size.

The rest of this article is organized into different sections. Section II discusses the scope and contributions of this article. Section III presents the broad spectrum of edge-AI and identifies EI as a vital element of edge-AI. Section IV puts forward an overview of bringing classical ML on edge. Section V explains DL methods, and Section VI includes techniques for edge-compatible DL model design and neural architecture search (NAS). Section VII is a review of compression techniques to make DL compatible with resource-constrained edge devices. Section VIII briefly explored the source, usage, and opportunities of sparse tensors in DL. Section IX identifies, lists, and discusses the EI performance evaluation metrics. Section X illustrates the popular software packages and techniques for codesign and optimization. A discussion on available hardware platforms and architecture optimization is presented in Section XI. Section XII outlined distributed edge training and inference. Section XIII presents a few use cases of EI. In Section XIV, challenges and future trends are analyzed, and conclusions are drawn in Section XV.

II. SCOPE OF THIS ARTICLE

Research on DL EI is flourishing as the theoretical foundations of DL algorithms are becoming rich. DL algorithms are now achieving human-level performance for many intelligent applications, such as voice recognition, behavior prediction, object tracking, autonomous driving, and visual recognition. These achievements have improved the quality of human life dramatically. People are more interested to have smart and intelligent services anywhere-anytime. Thus, there is a boost in research on bringing artificial intelligence (AI) services to the end users. Moreover, due to the rapid emergence of smart devices, mobile computing, sensors, and the Internet of Things (IoT), massive amounts of data are being generated at the edge. According to the IDC forecast, around 80 billion IoT devices and sensors will be associated with cyberspace by 2025 [3]. Transferring a tremendous volume of data from sensors at the edge to the cloud for processing creates a critical challenge to network capacity, transmission bandwidth, and computing

infrastructure. Besides, the chances of violating security and privacy increase during this transfer of data. Therefore, edge computing has emerged to meet the computing needs by distributed computing and bringing processing closer to the data sources. In addition, the improvement of edge computing systems facilitates the implementation of EI as a reality.

There are surveys available in various related areas. In [4] and [5], different DL theories, architectures, algorithms, and applications are described. Four prominent compression techniques, such as parameter pruning, low-rank approximation, compact CNN design, and knowledge distillation (KD), are summarized in [6]. This article lacks in including state-of-the-art techniques and discussing applications of compression on EI. Different compression techniques, such as pruning, weight sharing, knowledge sharing, and low-rank approximation, are reviewed in [7], which also includes the hardware acceleration of compression techniques. Reviews on hardware realization of convolutional NNs (CNNs) [8] in general and on the application-specific integrated circuit (ASIC) [9] have also been presented. These papers do not include software frameworks, codesign techniques, hardware platforms other than field-programmable gate arrays (FPGAs) and ASICs, and a complete pipeline associated with EI. Methods to obtain compact network architectures, such as joint compression, sparsification, tensor decomposition, and bit-adaptive computing, are surveyed in [10]. However, discussions on compression performance evaluation are limited, software frameworks are absent, and there is a lack of reviewing hardware acceleration platforms for EI. FPGA-based acceleration of DL tensor computation [11] and execution of compact networks on FPGA [12] are also studied. These two surveys limited their discussions to CNN inference on FPGA, which is only a small part of DL inference on edge. Efficient techniques, such as model compression and dataflow optimization for hardware accelerations of DNN, are explained in [2]. This tutorial and survey article mainly focused on designing custom neural accelerators and was limited to reviewing other hardware platforms, software packages, and DL techniques other than CNN. Different frameworks and compilers for ML are surveyed in [13] having no consideration of edge technology and compression-compiler codesign. Benchmarking quantized DL models to accelerate on smartphones is presented in [14]. Handling sparse and irregular tensors resulted in different DL models, and techniques in hardware are summarized in [15]. There are available reviews of different DL techniques [5], DNNs on edge [2], [16], DL compression techniques for hardware acceleration [10], DL inference on FPGA-based platforms [17], edge intelligence [18], and edge computing [19]. However, none of the reviews presents a complete picture of realizing DL inferences in edge devices. This article presents a complete pipeline and state-of-the-art techniques of DL inference on resource-constrained edge devices comprising model building, compression, codesign, software tools, hardware platforms, performance evaluation, and challenges.

However, with the rapid growth of DL concepts, these surveys lack a discussion of emerging trends and state-of-the-art tools and techniques. This article is an up-to-date survey on DL inference on edge devices. The training process contains complex calculations of derivatives and weight updates through backpropagation, which are not feasible for on-chip implementation in most cases. Therefore, the scope of this article is restricted to training on a different platform and inference on edge.

This review considers research on DL techniques, optimizations, algorithms, and architectures emphasizing EI. Therefore, the scope of this article is limited to lightweight DL models, different compression techniques, application areas, algorithm–hardware codesign approaches, available hardware and software tools, and use cases of EI.

The major contributions of this article are listed as follows.

1) Research achievements and enabling techniques of EI, such as compression, algorithm–hardware codesigns, and hardware accelerators, are discussed. A comprehensive classification of compression techniques, practical use cases, advantages, and limitations is provided.
2) Discussions and analyses of research papers in EI are presented from multiple viewpoints, such as the base network, the adopted optimization method, applications, and results. Key performance metrics to evaluate, compare, and benchmark the EI research are discussed.
3) The increasing importance and research trends of shifting DL inference from the cloud to resource-constrained edge hardware are highlighted. Available software tools for the hardware-algorithm codesign are reviewed. The available choices of edge hardware platforms are provided with the pros and cons analyses.
4) The design challenges and future opportunities for high-performance EI are explored. Open challenges in the realization of EI are identified, and a possible outline of future research directions is discussed.

III. ARTIFICIAL INTELLIGENCE ON EDGE

The edge-AI is the collaborative utilization of edge computing and AI. Big data generated on edge require AI to utilize its full potential. Besides, the availability of sufficient data paves the way for improving edge computing systems. This intersection between AI and edge computing has resulted in a broad application of edge-AI for precision medicine, agriculture, industrial IoT, visual and cognitive assistance, and smart home.

Fig. 3 illustrates the four essential elements of edge-AI [20]: 1) edge caching; 2) edge training; 3) EI; and 4) edge offloading.

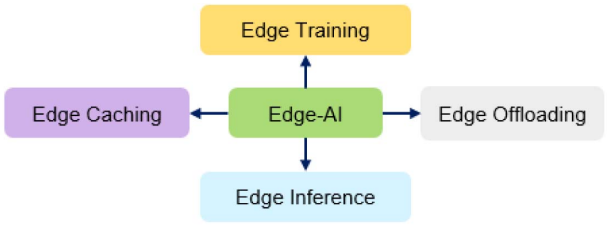


Fig. 3. *Four major elements of edge-AI are [20] edge caching, edge training, EI, and edge offloading.*

A. Edge Caching

Edge caching is the collection and storing of the data produced by edge devices, sensors, and IoT through a distributed and connected system. For example, images and videos captured by the smartphone and environment or patient monitoring data from sensors are collected and stored for edge-AI. Two types of data can be stored at the edge: 1) raw sensor data and 2) results from previous computations for reuse. Caching data on edge for reuse can substantially lessen the computational complexity and inference time. For instance, Cachier [21] caches the input features and corresponding task results on the edge server for reuse. The edge server transfers the least frequently used (LFS) input to the cloud. A 3× improvement of latency is possible using Cachier [21]. In Precog [22], the cached data are stored both at the edge server and mobile devices that achieve a 5× speedup. FoggyCache [23] uses caching to reduce the redundant computation and achieves 3× latency improvement and 10× reduction of energy cost. For redundant data handling, caching is an effective technique for faster inference on edge devices. CNNCache [24] reuses the results from similar image regions, thus minimizing computations and the burden on device resources.

B. Edge Training

Training on edge enables the devices to learn the patterns from the data cached on edge. Such training can take place either on an edge server or on an edge device itself. The edge training strategies include: 1) independent training and 2) collaborative training. Independent training is performed on a single-edge device without any support from the cloud. On the other hand, in collaborative training, highly computation-intensive training is performed on the cloud, and results are used by the edge, while part of the less computation-intensive training is performed on the edge device. Edge training is slower compared to training on the cloud. One of the challenges of collaborative training is maintaining the privacy and security of the data.

C. Edge Inference

EI is the computation of forward propagation of a trained ML model to get the decision from given inputs. The key enabling techniques are shown in Fig. 4. Most of the DL algorithms are designed to target higher accuracy by employing high-performance central processing units (CPUs), GPU, or multicore high-performance computing (HPC). Therefore, designing compact network architecture or compressing the existing models and ensuring real-time performance from the raw data are the two-core challenge of efficient EI. Designing new DL architecture targeting edge devices can be obtained in two ways: 1) design by experience and 2) NAS (see Section VI-A). For EI, design by experience is more suited than NAS due to the high computational needs of NAS algorithms. Model compression techniques are used to obtain acceptable performance with reduced model size in terms of parameters and computation. Similar to edge training, inference on edge devices is also slower than inference on the cloud.

DL inference can broadly be classified into two types: inference on cloud and inference on edge. Inference on the cloud is the traditional DL training and inference both on the cloud server or HPC data center. The collected data from edge devices and sensor nodes are sent to a centralized system to train the DL models and make the inference. This type is suitable for initial model building, training, and optimization. However, for providing service to the end user, EI is more suitable.

The EI of DL models can be categorized into three types [18]: 1) inference on the edge server; 2) inference on the edge device; and 3) collaborative inference on the edge device and edge server.

1) Inference on Edge Server: The input data collected from end nodes are sent to the servers at the network edge. The DNN model is stored, and inferences are performed at the edge server. The result obtained is then returned to the edge devices. As the DNN model remains available on the edge server, multiple edge devices for many applications can easily make the inference using the same model. However, the latency is dependent on the available bandwidth and communication resources between the edge server and the edge device.

2) Inference on Edge Device: In the inference on edge device mode, the DL model can be trained either in the cloud, edge server, or any other platform. The learned model is loaded to the edge device to make the inference locally. Privacy and security are maintained, and communication resources are saved in this method. The limitations are the resource constraints of edge hardware and the need for significant DL model compression to meet the performance goal. The focus of this article is restricted to the inference on edge devices.

3) Collaborative Inference on Edge Device and Edge Server: In collaborative inference employing both the edge device and the edge server, the DNN layers are partitioned into multiple parts. The edge device performs inference up to a partition point and sends the intermediate results to the edge server. The remaining part of the DNN model is

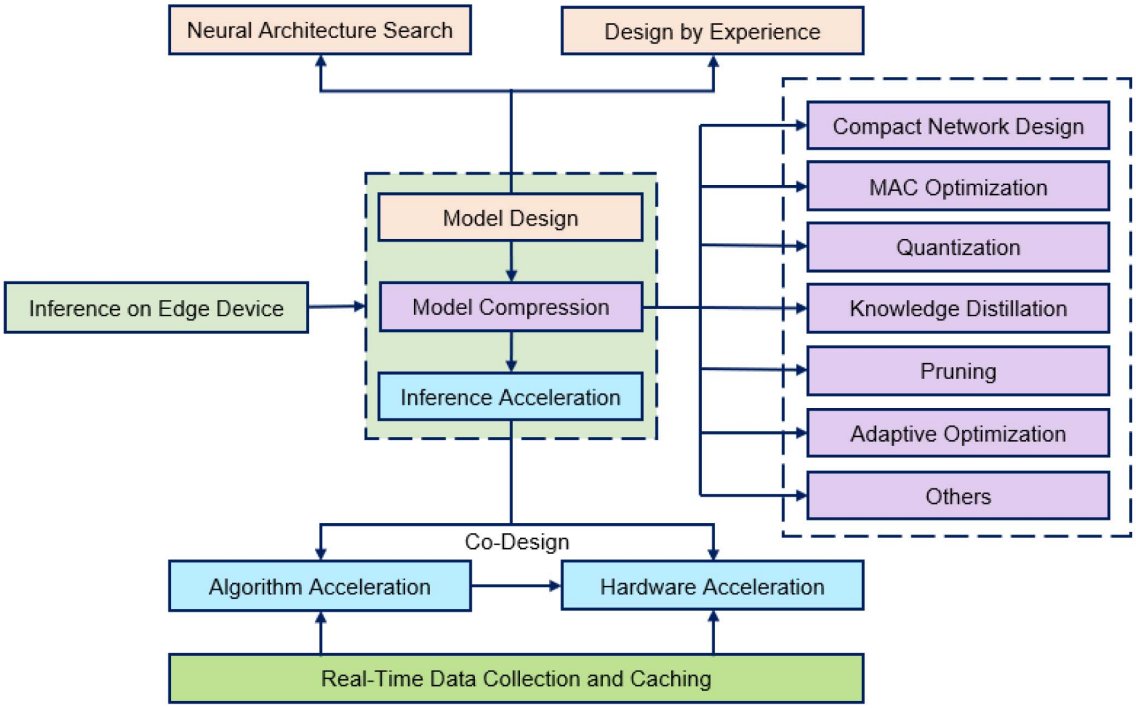


Fig. 4. *Key enabling techniques of EI. DL models can be designed by experience or NAS. The models are compressed using individual or joint compression. Hardware and software accelerations are used for inference on data collected and processed at the edge.*

executed on the edge server, and predictions are returned to the edge device. The collaborative edge device and server inference are more flexible than individual edge device or edge server-based inference. However, determining the partition point is challenging and requires additional computation.

D. Edge Offloading

Edge offloading provides computing support for caching, training, or inference depending on the needs. Edge offloading can carry out part of the processing in any of the connected distributed computing centers or cloud servers when the computing capability of edge hardware is not adequate. The performance of EI can be boosted by offloading the computation-intensive task to the cloud or by partitioning the task. For example, in [25], the latency and energy efficiency of each layer are estimated using a regression-based method to offload the computation-intensive task to the cloud to meet the latency and energy requirements. In [26], instead of uploading the entire model to the cloud, the layers are incrementally uploaded for collaborative inference, which significantly improves the energy cost and performance.

IV. CLASSICAL MACHINE LEARNING ON EDGE

In many real-world applications, ML algorithms [such as random forests (RFs), decision trees (DTs), support vector machines (SVMs), K-nearest neighbors (KNNs), and linear regression (LR)] are more appropriate than DL. Therefore, efficient ML inferences on edge devices are often crucial [27]. For instance, training and inference of ML models that use gradient calculation for optimization (such as SVM, K-means, and LR) are demonstrated in [28]. The training is performed on edge devices exploiting local gradient descent computation. Such techniques generate local copies of the ML model that are aggregated in a central edge device to calculate the weighted average of the local updates. The iterative process continues until reaching an optimal trained model within the resource budget. The performance was demonstrated by incorporating three Raspberry Pi's and a laptop connected through Wi-Fi. ProtoNN [29] is an implementation of KNN on edge devices demonstrating real-time inference and minimal storage (<2 kB of the Arduino UNO microcontroller). This technique uses a sparse projection matrix to project the data into lower dimensional space and uses a joint optimization to learn the projection matrix and prototype samples from data. Such joint optimization protects the accuracy degradation in binary and multiclass classification problems. ProtoNN technique is available in the Microsoft EdgeML library [30]. Tree-based ML techniques, such as RF and DTs, have logarithmic time complexity and linear space complexity to the training data size. Deploying such models on edge devices is complicated. Thus, often, either aggressive pruning or shallow tree models are used. Bonsai [31] maintains the accuracy of the DT by a low-dimensional linear projection of the input. This technique can be applied in a streaming mode without

storing the whole input in RAM. The low-dimensional projection and model parameters are jointly learned, which retains the accuracy. This technique has experimented with binary and multiclass classification problems requiring only 70 and 500 bytes for binary and 62-class classification, respectively, with comparable performance to the uncompressed version. Bonsai is also integrated and available as a part of EdgeML library packages [30].

V. DEEP LEARNING METHODS

An NN functions similar to neurons in the human brain. The neurons in NN map the inputs into activations for the next layer. The activations obtained by the multiply-and-accumulate (MAC) operations are fed as the input to the next layer and sequentially pass through the network architecture to generate the results. Conventional ML algorithms require handcrafted feature extraction from raw data that depends on domain expertise and signal processing knowledge. On the other hand, the key advantages of DL are automatic feature learning multilevel abstraction through different layers.

Model complexity, performance, maintainability, and resource utilization are a few criteria for selecting a model, which can be across different DL types or models from the same category tuned with various hyperparameters. Probabilistic measures and resampling methods are the two popular ways of selecting models. Probabilistic measures consider the model performance and complexity during the training stage [32]. Resampling methods mainly focus on the generalization performance during the development process. Resampling methods split the training dataset to create many training and test subsets. The DL model is trained and evaluated on each subset. Usually, this process is continued for multiple iterations, and the average performance over each test is estimated. Three popular resampling methods are random splitting [33], cross-validation [34], and bootstrapping.

The limitations of DL models to fit a particular data distribution are often characterized by bias and variance. A high-bias model cannot sufficiently capture the patterns in the data resulting in underfitting. High variance leads to overfitting the training data causing the model to perform very well on the training data but lacking generalization on test data. An estimate of the bias and variance can be obtained from learning curves. An optimal DL model can be selected by tuning the hyperparameters considering the bias and variance tradeoff.

Fig. 5 illustrates different training approaches commonly encountered in ML. Broadly, ML algorithms can be characterized by supervised and unsupervised learning. In supervised learning, the model is trained with the labeled dataset to predict the classes from unseen data. A loss function is incorporated that finds the difference between the prediction and the actual label. In the training phase, minimization of the loss function iteratively improves the weights and biases. The process of optimizing the loss function to update the model parameters is called training/learning. There are several optimization algorithms, such as stochastic gradient descent (SGD), ADAM, RMSProp, and AdaGrad [35]. In SGD, the average of gradients for several examples updates the weights in each step. The gradient calculation in DL is usually performed by the chain rule of derivatives starting from the output toward the input, termed backpropagation. In the evaluation phase, the trained model predicts from unseen data. Typical examples of supervised learning applications include weather forecasting, spam detection, face detection, sentiment analysis, pricing predictions, and so on [36]. Supervised ML techniques widely used are DTs, RF, naive Bayes, SVM, KNNs, LR, and NNs, among many others [37]. NN techniques are usually used in a supervised fashion but also find extensive applications in unsupervised learning. In unsupervised learning, models can learn without the labeled data. A few examples of unsupervised learning are handwritten digit recognition, speech recognition, natural language processing (NLP), DNA pattern analysis, and so on. Unsupervised learning techniques include K-means clustering, spectral clustering, principal component analysis (PCA), singular value decomposition (SVD), and so on [38]. Semisupervised learning combines supervised and unsupervised techniques in which training uses both labeled and unlabeled data. A few promising applications of semisupervised learning are speech analysis, drug discovery, web content classification, and banking, among many others [39]. Semisupervised techniques include expectation maximization with generative mixture models, transductive SVMs, graph-based methods, and so on [39], [40]. Reinforcement learning (RL) has exploded as a prominent technology that falls in the semisupervised learning category. RL does not explicitly require labeled input–output pairs for training and decision-making but rather learns from trial and error using feedback from its actions and experiences. Intelligent agents continuously learn to refine their actions over states from the environment to maximize the accumulative reward functions [41], [42]. The RL has numerous applications in robotic manipulation, the game industry, video streaming, NLP, and so on. RL algorithms include deep Q-network, Q-learning, double

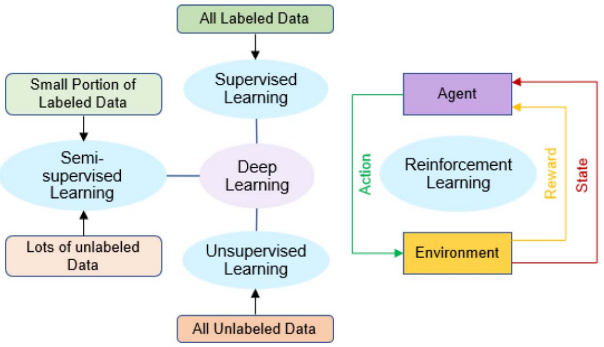


Fig. 5. *Different training schemes used in DL.*

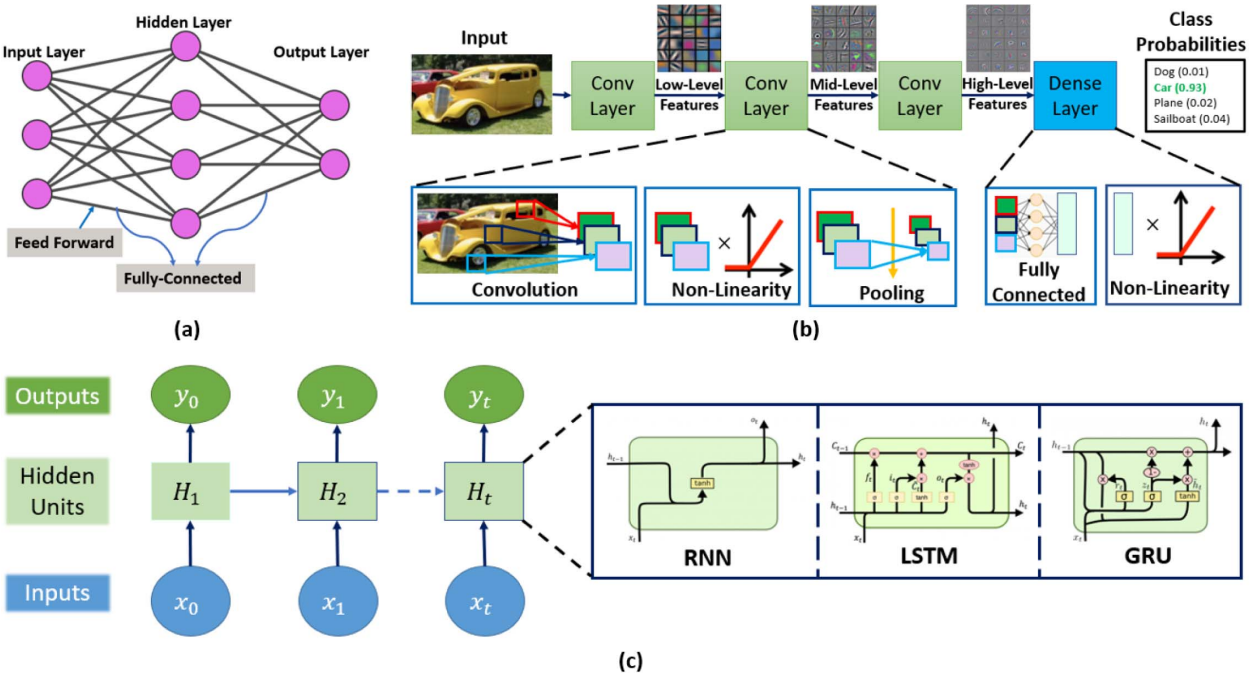


Fig. 6. *Three basic types of DL architecture: (a) FNN, (b) CNN [2], and (c) RNN. The FNN works as a classifier. In CNN, convolution operations are used to extract the features, and the pooling layer identifies and keeps the significant features through downsampling. In RNN, the memory in the hidden units keeps the previous result to improve the current output due to the long-term dependence on sequence data. The hidden unit H can be either vanilla RNN, LSTM, or GRU unit depending on the RNN type.*

Q-learning, actor–critic, and policy gradient, among many others [43].

Two common forms of DL architecture are feed-forward NNs (FNNs) and recurrent NNs (RNNs). In FNN, the computations are executed on the outputs of the prior layers. The network is memoryless, and the current output has no dependence on the previous input sequence. In contrast, RNNs, such as long short-term memories (LSTMs) and gated recurrent units (GRUs), have internal memory to incorporate long-term dependencies of sequential data. Intermediate results are stored internally in the network to be fed as inputs to subsequent layers. The RNNs are mostly used for time series and sequential data, such as speech and natural language analysis. The CNNs, mainly composed of convolution and pooling layers, are generally useful for computer vision and image processing problems due to local connectivity, shared weights, and the ability to use many layers. To improve the accuracy, the current trend is to increase the number of hidden layers in DL, which, in turn, increases the number of computations and parameters. To balance the increased cost by escalated depth, bulky convolution operations are being substituted by a sequence of light convolutions. Dropout is another technique most often used to reduce the model complexity by dropping some connections found in major applications in deep CNN (DCNN). As the dense layer is computationally more expensive with many weights, modern DL architectures typically contain a reduced number of dense layers. The three widely used DL models are shown in Fig. 6. The deep belief network (DBN), the autoencoder (AE), the restricted Boltzmann machine (RBM), the deep Boltzmann machine (DBM), the conditional random field (CRF), and the generative adversarial network (GAN) are a few examples of DL architectures that have evolved and adapted in many applications [5].

Usually, in DL, the output of each layer is connected to the input of the next layer. Some of the common nonlinear functions are rectified nonlinear unit (ReLU), leaky ReLU, sigmoid, Tanh, and ELU [44]. In multiclass classification problems, the softmax function is usually employed as the activation function of the output layer.

Over the last decade, DL has received widespread acceptance in many domains. Some of the applications are image classification, speech recognition, drug molecules activity prediction, restructuring brain circuits, DNA gene expression analysis, and disease diagnosis. Examples of DL applications in speech analysis and NLP include topic classification, keyword spotting, sentiment prediction, and language translation. DCNNs are prevalently being used for image processing and computer vision applications, such as object detection and tracking, traffic sign recognition, biomedical image segmentation, human detection, and face recognition. In addition, DL has paved the way for advanced driver-assistance systems (ADASs), autonomous mobile robots, and self-driving cars. Widespread acceptance and usefulness of DL have been triggered by the availability of huge training data, improvement of computational resources through parallelism, hardware and

Table 1 Computational Characteristics of Popular DL Models

DL Model	No. of Weights (M)	No. of MACs (G)	Network Depth	Memory (MB)
LeNet-5 [49]	$60k$	$341k$	5	-
AlexNet [50]	61	0.724	8	217
VGG 16 [51]	138	15.5	16	512.2
GoogleNet [52]	7	1.43	22	40
ResNet 50 [53]	25.5	3.9	50	97.70
ResNet 152 [53]	60.2	11.3	152	230
Inception v3 [54]	23	5	48	91
Inception v4 [55]	42.7	12.3	77	163
Xception [56]	23	8.42	38	-
DenseNet161 [57]	28	7.82	161	-
SeNet154 [58]	115	20.82	154	-
NasNet-L [59]	89	24.04	29	-

software optimization for fast and reliable simulation, and evolution of different variants of new and efficient DL model architectures. However, the computational and memory needs for DL models are evident from Table 1. With the recent advancement of DL algorithms, accuracy has significantly improved with deeper layers at the cost of increased computational and memory needs. For instance, the number of parameters increased from 50 to 100 million to 175 billion for NLP tasks using the Transformer [45] network to the GPT-3 [46] model within a gap of only three years (2017–2020). Since 2012, DL has experienced a 3000× and 10× increase in computational need and memory requirements, respectively [9]. Studies demonstrate that these models are often overparameterized having significant redundancy [47], [48]. Networks having millions of parameters and computations are not feasible to fit in edge devices without significant optimizations. Thus, compression techniques discussed in Section VII can be used to obtain a compact network architecture with reduced computation complexity, execution time, and minimal memory footprint.

VI. EDGE-COMPATIBLE DEEP LEARNING MODEL DESIGN

Larger DL models with big data can generate better results, which are often not sustainable due to resource limitations at the edge and the requirement of low inference time to ensure real-time applicability. Therefore, instead of creating deeper models, the trend has shifted toward a resource–performance tradeoff designing reasonable models feasible for edge deployment with acceptable performance. Such edge-compatible models can be designed either automatically using techniques, such as NAS or designing compact models exploiting research experience. This section is a brief on edge-compatible DL model designs in two ways: 1) NAS and 2) compact network design.

A. Neural Architecture Search

DNNs for resource-constrained edge hardware devices require compact computations and a reduced number of parameters while preserving acceptable accuracy. Appropriate network architecture can be obtained after the evaluation of many architectures that are often time-consuming, tedious, and have an infinitely large search window. The NAS is an automatic and algorithmic way of finding optimal NN architectures and components.

The basic NAS strategy is illustrated in Fig. 7. NAS techniques have three parts [60]: search space, search strategy, and performance estimation. Search space is the available architectures discoverable in the search. To limit the search window, a search space is defined by imposing constraints on network architectures, operations, layers, repeated patterns, and so on. Although these constraints introduce human bias to the NAS, the search space will remain sufficiently large to discover novel architectures for a particular problem. Search strategy determines the guide to explore the search space. Search strategy can be random, but it often considers the candidate architectures based on the performance of previously explored working solutions. Agents can be trained through RL [61] to estimate the performance of a novel architecture on unseen data. There are techniques to estimate performance through parameter sharing instead of training from scratch and using approximated data and networks.

Software tools, such as Google Cloud AutoML [62] and Auto-Keras [63], can be exploited for NAS. AutoML can take the training data as input and outputs the optimal building blocks for a base network construction that are fine-tuned for further optimization. In Auto-Keras, the algorithm searches for the detailed configuration of the user-specified high-level architecture.

MobileNet-v3 [64] released in large and small versions uses NAS and NetAdapt [65] algorithms. The Netadapt [65] algorithm is presented in Fig. 8, which is an adaptive algorithm to generate many network proposals based on given quality metric specifications. The algorithm evaluates the performance of these proposal models based on experiments on selected hardware and selects the one with the highest accuracy. Compared to MobileNet-v2, the large MobileNet-v3 results in 25% speedup with comparable accuracy, and the smaller version offers 6.6% increased accuracy with similar size and delay. NASNet [59] starts with a sequence of blocks in which the construction of blocks is determined by RL. NASNet consists of the normal block and the reduction block. The convolution

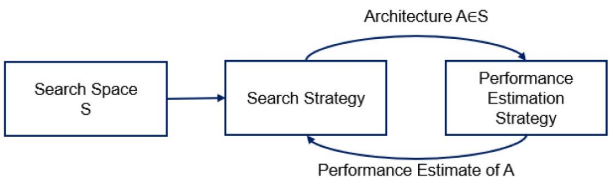


Fig. 7. *Components of NAS. NAS techniques consist of a search space S. Using the search strategy an architecture A is obtained, which is evaluated using the performance estimation strategy.*

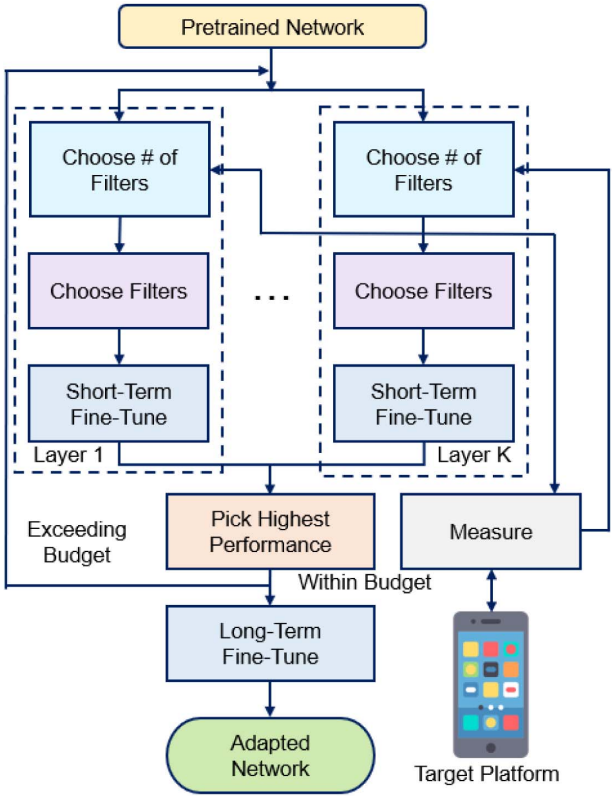


Fig. 8. *NetAdapt [65] algorithm. Layers are simplified at each iteration based on empirical measurements. The network having maximum accuracy within the target platform's budget is chosen for fine-tuning.*

in the normal block keeps the size of the feature map, which is reduced by 4× in the reduction block. Accuracy improvement of 3% can be achieved in NASNet compared to the MobileNet with a similar number of parameters. Progressive NAS (PNAS) is reported in [66] for CNN architecture search. PNASNet demonstrates 8× faster convergence and 5× improved efficiency than RL-based model search. MNASNet [67] automatically generates DL models for mobile platforms. RL is used to explore and find the architecture constituents from a predefined search space that considers a tradeoff between accuracy and mobile device constraints. On mobile devices, MNASNet can classify images with 78-ms latency, which is 1.8× faster than MobileNet-v2 and 2.3× faster than NASNet with 1.2% higher accuracy. The accuracy is 0.8% and 1.2% better than MobileNet-v2 and NASNet, respectively.

Hardware-aware transformer designs using NAS techniques for edge-compatible and low-latency inference are presented in [68]. The search space consists of numerous encoder–decoder attention and heterogenous layers. A SuperTransformer is trained to generate many SubTransformers, and through hardware exploration, the faster SubTransformer is identified. For machine translation, NAS technique-based transformer model exploration resulted in 3× speedup and 3.7× reduced size on a Raspberry Pi-4 device.

The NAS algorithms are computationally expensive. To minimize the hardware dependence of NAS, differential architecture search (DARTS) [69] has evolved. DARTS techniques use gradient descent for architecture search, and both CNN and RNN models can be obtained. However, NAS techniques are still in the infancy to be adopted to find compact architectures for EI.

B. Compact Network Design

1) Optimized Convolution in CNN: In CNN, most computation time and power are consumed by the convolution operations. The faster inference of a CNN mostly depends on how fast a single input can be processed. Designing compact network architectures can minimize the number of weights and computations. For instance, concatenating a series of smaller filters can emulate the same effect of a larger filter. Convolutions are simplified using 1×1 convolution in MobileNet [70] and Xception [56] networks. There are two major optimizations of the convolution operation in MobileNet: input size reduction and pointwise convolution. The depthwise separable convolutions are used, which are a combination of depthwise convolution and a pointwise convolution. The network structure is a sequential convolution subsequently average pooling and, finally, a dense layer with configurable hyperparameters. A depth multiplier is used that can change the number of filters in each layer. For a variety of applications, MobileNet with 0.5–4.2 million parameters and 41–559 million MACs achieves an accuracy between 50% and 70%.

A hybrid DL model called EdgeNeXt is presented in [71], which combines the power of CNN with the self-attention mechanism of a transformer for computer vision tasks considering the resource constraints of edge devices. EdgeNeXt, with around 28% reduced floating-point operations (FLOPs), achieves 71.2% top-one accuracy on the ImageNet dataset for classification having only 1.3M parameters.

Fig. 9 shows different types of convolution operations used to obtain compact CNN models. In MobileNet-v2 [72], residual connections and intermediate data encoding have been introduced to reduce the number of operations and weights. The elementary unit of MobileNet-v2 is the bottleneck residual block consisting of three convolutional layers. The three convolution layers are one 1×1 convolution followed by two depthwise separable convolutions. MobileNet-v2 reduces about 30% parameters and 50% operations compared to MobileNet with improved accuracy. 1×1 convolution is also used in SqueezeNet [73] to reduce the number of channels in a CNN. SqueezeNet performs three main optimizations: 1) reduction of the number of inputs of filters; 2) replacement of 3×3 filters by 1×1 filter; and 3) downsampling with global average pooling. Inputs to the large filters are

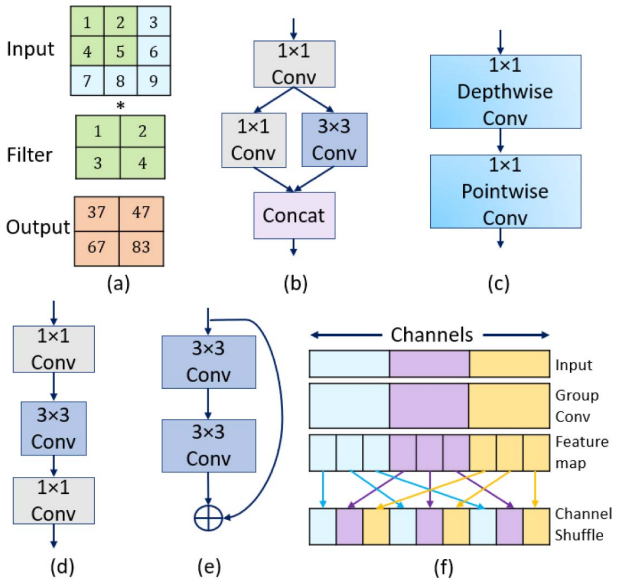


Fig. 9. ***Different types of the convolution operation. (a) Basic convolution. (b) Fire module. (c) Depthwise separable convolution. (d) Bottleneck layer. (e) Residual connection. (f) Channel shuffle [20].***

first squeezed by 1×1 convolution and then expanded by a set of parallel 1×1 and 3×3 convolution filters. This squeeze and expansion operation block is called the fire module. The SqueezeNet consists of an initial convolution layer, eight fire modules, and a global average pooling layer at the end. The result is a 50× reduction in the number of weights than AlexNet while preserving the equivalent accuracy. However, one drawback of SqueezeNet is that it consumes more energy than AlexNet. Instead of using a fire module, a two-stage bottleneck module is used in SqueezeNext [74]. 3×3 convolutions are substituted by separable convolutions and 1×1 expansion modules afterward. SqueezeNext has shown 112× fewer parameters compared to AlexNet but resulting in better accuracy.

In ShuffleNet [75], convolutions are applied parallelly to separate parts of the inputs reducing the number of operations. The 1×1 convolutions are replaced by group convolutions and a channel shuffle forming the ShuffleNet unit. The output of convolutions is shuffled to mix the information from different groups. ShuffleNet has shown promising results with 3% better accuracy compared to MobileNet-v1 with similar computational complexity. In ShuffleNet Sx, a stride of two is used to implement 3×3 convolutions. Elementwise additions are replaced by concatenating channels. The hyperparameter s is used as a scaling factor for the number of channels. The overall network architecture consists of an initial convolutional, the average pooling layer, and three ShuffleNet units afterward. Instead of using random shuffling, CondenseNet [76] learns the grouping during training. It also integrates the parameter pruning and removal of less important features. CondenseNet with 50% fewer parameters can achieve ShuffleNet-like accuracy. ANTNet [77] is an image classification CNN that also utilized the group convolution concept. This network reported 0.8% increased accuracy, 6% parameter reduction, 10% fewer operations, and 20% performance boost compared to MobileNet-v2 for inference on mobile devices.

The Winograd algorithm [78] can significantly reduce the computation time of convolution with small filters, minibatches, and minimal arithmetic convolution over small tiles. The number of floating-point multiplications and integer arithmetic operations can be reduced by a factor of 2.25× [79] and 3.13× [80], respectively, using the Winograd algorithm. The Winograd algorithm can minimize computations in the convolution exploiting additions and shifts. As additions can be implemented in hardware with negligible power consumption, Winograd convolution demonstrated potential for hardware inference. FPGA implementations of Winograd convolution are presented in [81], which incorporate feature map caching using linebuff structure, data reuse, effective use of pipelining for PEs, and parallel processing of convolution operations. A unified architecture incorporating the Winograd and general matrix multiplication (GEMM) named UniWig is presented in [82]. Instead of using separate PEs for convolution and dense layers, UniWig utilizes the same set of PEs and blocked Winograd filtering to ensure proper resource utilization. Parameters in convolutional layers remain unchanged over a long interval in weight-stationary CNN. Thus, the CNN layer can exploit data reuse through a reconfigurable convolutional kernel [83]. The reconfigurable architecture efficiently utilizes the lookup tables (LUTs) of FPGA for reducing computational complexity.

2) Optimized RNN Design: Designing compact RNNs can be realized either at the unit level or the network level. The LSTM and GRU are the unit-level improvements of the vanilla RNN by eliminating the vanishing or exploding gradients. Studies illustrate that, in LSTM, the forget get and the input gate plays the most critical role, and the output gate is less significant [84]. Therefore, in GRU, the input and forget gates are coupled into the update gate resulting in a simplification of LSTM. The units of S-LSTM [85] and JANET [86] consist of only one forget gate. The minimal gated unit (MGU) integrates the reset gate and update gate of the GRU [87]. LSTM can also be extended by adding an extra time gate for rapid convergence and performance boost [88]. Such unit-level modifications resulted in simplified RNN architecture having reduced model complexity without performance degradation. Faster results can be obtained using the phased LSTM [88], which extends the LSTM unit cell by adding an extra gate called the time gate.

The network-level optimization pays attention to the interaction between different RNN or LSTM cells instead of the network units. The network-level compression can be performed by modifying the stacked RNN layers in the temporal and spatial directions. For instance, the dimension of the hidden state can be minimized using the linear

recurrent projection layer in LSTM to improve the parameter efficiency [89]. The weight matrix of LSTM can be factorized into matrices, and results can be concatenated for faster convergence and parameter reduction [90]. Residual connections [91] and skip connections [92] can also be exploited in RNN for faster convergence and performance boost.

The concept of weight sharing is utilized in the grid LSTM [93] for multidimensional data processing. Parameter sharing is also exploited in the RNN and LSTM [94] by increasing tensor size with shared parameters. Thus, the network can be effectively widened without adding more parameters.

VII. DEEP LEARNING MODEL COMPRESSION FOR EDGE INFERENCE

Compact models and efficient compression techniques are developed to meet the massive trends in DL migration from the cloud to the edge. Instead of highly accurate models, the tradeoffs among performance, silicon area, cost, and power consumption are the key to EI. This section covers compression techniques to get optimized DL models compatible with resource-constrained edge devices.

A. Pruning

Most of the modern DL models are overparameterized. Pruning is one of the compression techniques that remove less important weights (connections) or filters (channels) from a trained model. The advantages of pruning are mainly twofold: minimizing the physical size of parameters and reducing the inference time. Pruning can be either structured or unstructured [95]. Unstructured pruning removes the weight and preserves the neurons if at least one connection to that neuron exists. A sparsification technique for LSTM network implementation on FPGA has been demonstrated in [96]. It splits the matrix into multiple banks containing an equal number of nonzero elements. The accuracy can be maximally preserved by keeping relatively large weights. Unstructured pruning results in irregular computation and memory access that limit parallelism for hardware implementations [96], [97]. In structured pruning [98], a group of weights is removed instead of pruning individual weights. The group of weights can be neighboring weights, part of a filter, or the whole filter [99]. A structured pruning technique presented in [100] for CNN removes certain filters to increase the efficiency of the network without significantly affecting the accuracy. An agent is trained layer-by-layer for pruning. The network performance is evaluated by retraining, and subsequently, the process continues for the next layers. Structured pruning offers efficient processing in hardware by exploiting the data-parallel architecture [101], improves network compression, and reduces storage requirements.

Structured and unstructured pruning can be combined to obtain an optimized model for hardware realization. For example, in [95], in each iteration, a small percentage of the least important weights based on a predefined threshold are removed from every layer, and the models are retrained to evaluate the performance. After the weight removal, if all connections are deleted from a neuron, that neuron is also removed.

Weight pruning can compress the weights of shallow networks without any significant loss of accuracy. An iterative vectorwise sparsification is presented in [97] for CNNs and RNNs. Result demonstrates faster response with a negligible accuracy loss compared to a 75% sparse network. However, filter (channel) pruning introduces a significant loss of accuracy for deeper networks.

Fig. 10 demonstrates different pruning techniques and effects on the network architecture. Currently, there are three main approaches to pruning to make sparse networks.

1) Magnitude-Based: The contribution of weight is proportional to its absolute value. Weights less than a threshold are removed to make the network sparse. Magnitude-based sparsity can increase the speed of convergence. In [47], a magnitude-based pruning is applied layer by layer. The model is retrained after pruning all layers to compensate for the effects of weight removal. The remaining weights are adjusted over several iterations. Pruning demonstrates 9× and 3× reductions of the number of weights and MACs, respectively, in AlexNet. Also, 9.9× weight reduction in dense layers compared to 2.7× for convolutional layers can be achieved. Empirical results from AlexNet and VGGNet illustrate that around 80% and 50% weights can be removed with and without fine-tuning.

A dynamic pruning technique is reported in [102] to minimize the inference latency of transformer networks while deploying on edge devices. The threshold-based pruning techniques have been demonstrated to reduce MAC operations and tensor data sizes while maintaining around 98.4% accuracy for keyword spotting tasks. Moreover, for an accuracy degradation tolerance of up to 4%, it can achieve up to 94% reduced operations and multihead self-attention inference speedup up to 16× compared to the keyword transformer. ABERT [103] is a pruning framework for automatic and efficient pruning of the BERT network to find an ideal candidate with high accuracy with given pruning ratio constraints. Experimental results on deploying the subnetwork inference on Xilinx Alveo U200 FPGA have shown 1.83× speed compared to the BERT model.

One limitation of magnitude-based pruning is identifying the appropriate threshold. In addition, if the threshold is shared among all layers, the result is not optimal due to the variation of the magnitude of parameters across different layers. Finding different thresholds for each layer is even more challenging. To overcome this limitation, differentiable pruning [104] can be used. A criterion has been set in [105] based on the Tylor expansion to identify less significant neurons in CNN. In [105], pruning

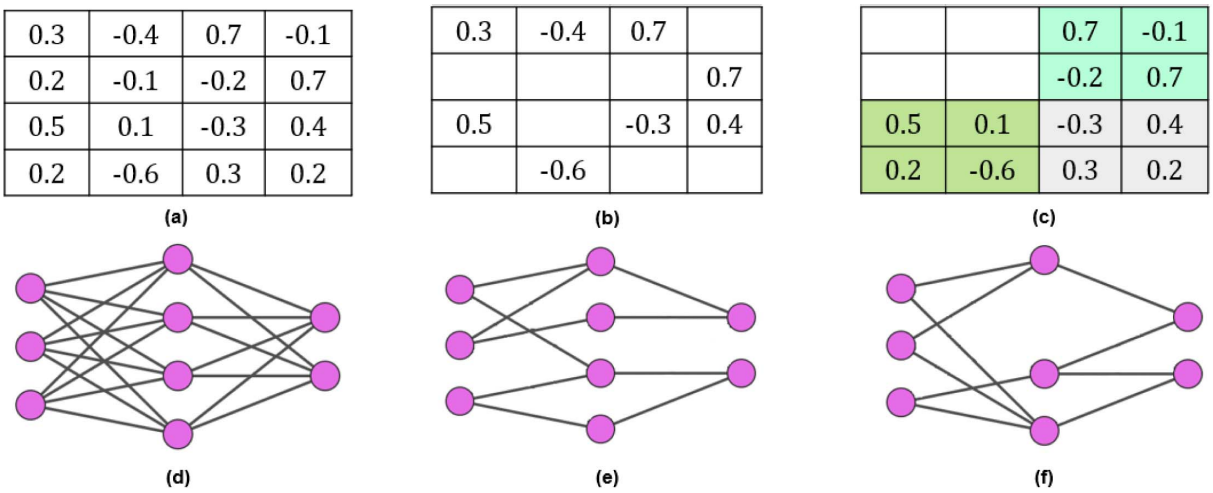


Fig. 10. *Illustration of pruning techniques. (a) Original weight matrix. (b) Magnitude-based unstructured pruning of 50% smaller magnitude weights. (c) Structured block pruning (pruning based on the lowest average of 2 × 2 weight groups). (d) Basic FNN architecture without pruning. (e) Result of unstructured 50% weight pruning on (d). Example of neuron pruning on (d).*

is formulated as an optimization problem to determine the cost minimization weights and allows simultaneous training of the network and pruning parameters. Gate decorator [106] is a global filter pruning algorithm for CNN based on scaling the channels. If the scaling factor is zero for any filter, that filter would be removed.

2) Regularization-Based: These methods use a regularization part in the loss function equation to provide the required sparsity level [107]. The network converges to a sparse subnetwork that has the desired sparsity rate and minimal accuracy drop. The main advantage is that the regularization-based method provides better accuracy as it does not corrupt the weights. However, regularization-based methods need a lot of iteration for convergence.

3) Energy-Aware Pruning: This technique is developed in [108] that prunes weights depending on an estimated energy. The energy estimation considers the number of MACs, data sparsity, and movement in the memory hierarchy. Energy-aware pruning demonstrated 1.74× more energy efficiency for AlexNet compared to magnitude-based pruning.

One of the disadvantages of unstructured pruning is a sparse weight matrix that complicates hardware realization. Pruning is usually more efficient for dense [fully connected (FC)] layers than other layers. Also, it does not necessarily reduce the latency because of the sporadic erasure of weights, making matrix multiplication slower. Other potential drawbacks of the pruning technique are lack of generalization for various architectures, irregular access to on-chip memory, the requirement of fine-tuning to achieve acceptable accuracy, and unbalanced parallelism in the computation.

Zero-skipping techniques [34] can avoid the multiplication by zero resulting from pruning. Skipping zeros from weights and activations may result in reduced performance efficiency [109]. The large on-chip memory is required in zero-skipping techniques to exploit the parallel processing in hardware acceleration. To minimize on-chip memory requirement, the weight matrix can be stored in a dense format [110]. The map batching technique balances the heavy transfer of parameters in dense layers by kernel reuse eliminating the transfer to an external memory [111]. Moreover, if filters (channels) are pruned, the model accuracy usually degrades. For this reason, CNN models are usually compressed with unstructured pruning to obtain a new network of the same size having sparse weight tensors.

B. Quantization

Weights and activations in most DL models use a floating-point representation that retains information but leads to slow processing. DNNs are trained to capture the key patterns and features in the data and are resilient to noise. Quantization maps the model parameters and activations into low-precision quantized levels avoiding costly FLOPs. Due to low precision, the number of bits is reduced, allowing more efficient integer arithmetic operations on hardware. DL can tolerate small changes in weights and biases resulting from quantization errors. Quantization offers three major benefits: 1) memory saving due to compact representation with fewer bits; 2) reduced complexity of mathematical operations resulting in improved inference time; and 3) improved energy efficiency.

There are two main ways of mapping data into different quantization levels: 1) linear and 2) nonlinear quantization. The quantization bit length can be fixed or variable. In fixed quantization, the same quantization is applied for all layers, data, and filters in the model. In variable quantization, different quantization can be used for different

layers, filters/channels, weights, and activations in the network.

Instead of using single-precision 32-bit floating point (FP32), the calculations and implementation of DL algorithms become much easier in 16-bit floating-point [112], 8-bit floating-point [113], and fixed-point representations [114], [115], [116], [117]. For example, an 8-bit integer (INT8) addition results in a 3.3× and 3.8× reduction in energy and area, respectively, than a 32-bit fixed point representation [118]. The same INT8 addition can save 30× energy and 116× area compared to FP32 operations. Similarly, 8-bit fixed-point multiplication consumes 15.5× less energy and 12.4× less area than the 32-bit fixed-point operations. An 18.5× less energy and 27.4× less area is required than that of FP32 multiplications. Therefore, the energy and area reduce approximately linearly and quadratically for fixed-point additions and multiplications, respectively, with the number of bits. AlexNet can be quantized using a variable precision of 4 to 9 bits with less than 1% accuracy degradation [119]. In [120], activations and weights of intermediate layers are quantized to 8-bit integers, whereas the inputs and biases remain in FP32s. Result demonstrates a 3–4× reduction in memory and 10× speedup without affecting accuracy. Quantization of only weights to 8-bit integer and output to 16-bit integer can result in 4.16× storage reduction [121]. Instead of floating-point numbers, posit numbers can be used to reduce storage. As posit numbers require fewer bits than floating-point numbers, it saves memory. For example, in [122], CNN is compressed by using posit numbers to read and write weights from/to memory. As the conversion of parameters occurs between float and posit, no quantization happens, thereby eliminating the need for retraining.

In linear quantization, the floating-point numbers are represented with fixed-point numbers. Scaling and biasing are used to minimize the quantization error caused by the difference between floating-point numbers to the nearest quantized fixed-point numbers. Mixed-precision quantization can be adopted, which applies different bit widths at different layers. Such flexible bit width can ensure a gain in memory efficiency and reduced power consumption. Stripes [123] is an implementation of variable bit width with bit-serial computation. It is a hybrid architecture that provides flexible bit width for the activations and fixed-point integers for weights. One potential problem of variable bit width is the increased latency, which can be minimized by exploiting the inherent parallelism of DNN models. The bit-serial operation uses AND gates and adders, eliminating the need for multipliers. Multipliers are one of the most power-consuming elements in DL processing. By omitting the need for a multiplier, it also excludes external memory access. UNPU [124] also implements bit-serial computation with 16-bit fixed-point representation for activations and 1–16-bit variable bit width for weights. Loom architecture exploits bit-serial multiplication and variable bit width for both weights and activations [125]. For efficient processing, Loom requires transposing of the output, which increases the overhead. The PEs in the bit fusion [126] architecture are combined differently to implement spatial flexibility in bit widths. The multiplication is partitioned into two 2-bit × 2-bit multiplication followed by a shifter and adder. The BitBlade [127] follows a similar architecture that eliminates the shift-add operation using bitwise summation, resulting in improved optimization. Often, low bit width is used for middle layer weights, and large bit-width quantization is used for weights in the first and last layers [128]. Weights and activations can also be grouped to use different bit widths to encode values of different groups [129]. A summary of different quantization techniques, quantized bit width, and neural accelerators is presented in Table 2.

Binarization is an extreme type of quantization that uses a 1-bit representation for weights and activations. For instance, binary values are used for weights in YodaNN [130], and both the weights and activations adopt binary values in BRein [131]. Binarization significantly speeds up the NNs by replacing the original FP32 multiplication with a 1-bit logical XNOR operation. The concept of binarization was introduced in BinaryConnect (BC) [132] to use binary weights (-1 and $+1$) that reduce the MAC to only additions and subtractions. BC [133] randomly binarizes the weights. Then, quantization is applied layer-wise to replace the remaining multiplication with bit-shift operations. No loss in accuracy was encountered during the training but sometimes achieves improved results. Binarization reduced the multiplication to addition and convolution to addition and subtraction operation reducing the size of the network approximately by 32× [134] and achieving 2× speedup. Binary weights and activations can also be applied in backward propagation to enhance performance through expectation backpropagation (EBP) techniques [135]. A fully binary NN (BNN) is presented in [136] using EBP demonstrating improved energy efficiency on the neuromorphic chip.

The major challenge of a BNN is to maintain the model accuracy. For example, BC [132] and BNN [137] result in an unacceptable accuracy degradation of 19% and 29.8%, respectively. Another limitation of BNN is that it can achieve acceptable performance only when datasets are small. The BNN model accuracy can be improved in two ways: 1) increasing the number of weights and 2) adopting several binarizations to input activations (IAs) [138]. Bi-Real Net [139] is proposed to minimize the accuracy degradation of 1-bit CNN incorporating real activations after a 1-bit convolution operation. To mitigate the problem of accuracy degradation, the binary weight nets (BWNs) and XNOR-Net [140] introduced a few modifications, such as scaling the output to recover the dynamic range, keeping the input and output layer with FP32 representation, and normalization prior to convolution. The BWN was able to limit the accuracy degradation to 0.8% and XNOR-Net to 11%. In XNOR-Net [140], the weight and convolution inputs are binarized. The convolutions are

Table 2 Summary of Noteworthy Works on the Quantization of DL Models

Quantization Technique	Model/Method	Bit-Width (no. of bit)	
		Weights	Activations
Fixed-point precision	Weight fine tuning [114]	8	8
Reduced precision for weights	BinaryConnect (BC) [132]	1	32-bit floating point
	Ternary Weight Network (TWN) [145]	2 (32-bit float for first and last layer)	
	Trained Ternary Quantization (TTQ) [146]		
Reduced precision for weight and activation	XNOR-Net [140]	1 (32-bit float for first and last layer)	
	Binarynet [137]	1	
	DoReFa-Net [143]	1 (32-bit float for first and last layer)	2 (32-bit float for first and last layer)
	Quantized Neural Network (QNN) [142]	1	
	HWGQ-Net [144]	1 (32-bit float for first and last layer)	
Nonlinear quantization	Incremental Network Quantization (INQ) [152]	5	32-bit floating point
	DeepCompression [157]	8 (conv), 4(dense)	16
		4(conv), 2(dense)	16
Variable precision neural accelerators	Stripes [123]	16	1-16
	UNPU [124]	1-16	16
	Loom [125]	1-16	1-16
	BitFusion [126]	1,2,4,8,16	1,2,4,8,16
	BitBlade [127]	1,2,4,8,16	1,2,4,8,16

simplified using XNOR operations, resulting in 58× speedup. The accuracy degradation of XNOR-Net can be improved by introducing 2-bit quantization of the activations instead of using 1-bit while keeping the weights binary [141]. Quantized NNs (QNNs) [142], DoReFa-Net [143], and HWGQ-Net [144] are a few examples of DL models that use 2-bits and 1-bit precision for activations and weights, respectively. In ternary weight nets (TWNs) [145] and trained ternary quantization (TTQ) [146], ternary weights instead of binary weights are used with 32-bit floating-point representation for activations that exhibit as low as 0.6% accuracy degradation. TrueNorth chip [147] also implemented the ternary weights and binary activations.

Nonlinear quantization considers the nonuniformity of weights and activations. An LUT containing the binary codes for weights and activations can be used. For example, replacing the multiplication operation with an LUT in [148] or using a hash function to create and train values from the LUT [149] has already been proposed. In [149], hash functions are used to uniformly cluster the weights in different hash buckets. Activation functions can be stored in the LUTs to speed up the activation calculation. Log quantization and weight sharing are two nonlinear quantization techniques. In log quantization, the quantization levels are determined by a logarithmic distribution that ensures equilibrium of weight and activation distribution at different quantization levels, reducing the quantization error. For example, log base-2 quantization has shown 5% accuracy degradation compared to 27.8% accuracy degradation for 4-bit precision linear quantization [150]. Quantization of weights as the nearest power of two can replace the multiplication with faster arithmetic shift operation that significantly reduces the computation [114], [151]. Incremental network quantization (INQ) [152] divides the weights into different groups by separating large and small values, and applies iterative quantization and retraining to reduce the accuracy degradation.

Temporal dependence of the RNN models accumulates the quantization error resulting in accuracy degradation. Though most of the quantization methods for RNN are like CNN, there are research efforts to quantize the RNN model itself. For instance, improved accuracy has been achieved in [153] using the binary quantization to RNN. A hybrid quantization method has been proposed in [154] for RNN that used threshold for weights and probabilistic methods for the hidden states' quantization. However, most of the RNN quantization methods used small datasets. Besides, there is a lack of benchmarks to evaluate the performance of RNN quantization.

Quantization-based compression is also effective for automatic speech recognition tasks on edge devices. For instance, exploiting 8-bit fixed point precision for weights and activations, SpeechTransformer [155] achieves a 4× reduced number of parameters while maintaining the performance of a full-precision model. An integer-only quantization of vision transformers for computer vision tasks is reported in [156]. Such integer-type quantization allows realizing the inference operation through integer arithmetic and bit shifting, achieving up to 4.11× inference speedup.

Some of the challenges of quantization include information loss due to low bit width, distorted network architecture, and complicated differentiation in backpropagation.

C. Joint Compression and Weight Sharing

Pruning and quantization techniques are applied either individually or jointly. Quantization and pruning techniques are combined in [157]. The less significant connections are first pruned to minimize the computations.

After pruning, the remaining parameters are quantized to reduce the storage requirements. The network is retrained to offset the effects of compression. The last step is the Huffman prefix coding to further compress the network and reduce the storage. In [158], pruning methods are implemented for popular DL models targeting IoT and mobile devices. A library for ARM Cortex-M processors named CMSISNN is presented in [159] for maximized performance through quantization. A combination of pruning and quantization has been demonstrated in [160] for RNN models. The result shows a 10× and 2× speedup through pruning and quantization, respectively. To meet the resource constraints of edge devices, Adadeep [161] presents an automatic selection of compression techniques and filters. DeepMon [162] uses a combination of quantization and caching to speed up the execution by reducing redundant computations of video frames. CPU and GPU can significantly increase the throughput by exploiting parallelism using 8-bit quantization. Instead of one 32-bit operation, four 8-bit operations are possible within one clock cycle with a faster response. For instance, the Google Tensor Processing Unit (TPU) and NVIDIA PASCAL GPU support 8-bit integer arithmetic operations.

In weight sharing, the same weights are used to calculate outputs for multiple layers or filters reducing the number of unique weights [157]. In [157], weight sharing is performed by k-means clustering and Huffman coding. As the same weights are repeatedly used, the memory requirements are significantly reduced. A hashing function can be used to group the weights and then use one weight from each group [149]. The number of parameters in CNN can be reduced by using vector quantization [163]. The concept of vector quantization is to use a representative central value for a group of parameters. K-means clustering is used to group the parameters, and then, vector quantization is applied to each cluster resulting in 16–24× parameter reduction within 1% accuracy degradation. In deep compression [157], the number of distinct weights is grouped resulting in only 8- and 4-bit precision weight indices in AlexNet for convolution and dense layers, respectively. Weight fetching becomes a two-step procedure: reading the weight index and reading the shared weight based on the weight index. Weight sharing is only useful for memory saving and does not impact the precision of MAC or computation.

D. Multiply-and-Accumulate Optimization

One of the main computational complexities of DL algorithms comes from multiplication operations. Therefore, optimization of the multiplication by replacing it or removing part of the multiplication can significantly improve the inference performance. For FPGA implementations, spatial convolutions are replaced by shift operations and depth convolutions followed by 1×1 convolutions in DiracDeltaNet [164]. Results demonstrated a 48× and 65× reduced parameters and operations, respectively, of the VGG-16 model having a comparable performance on the ImageNet dataset. The technique presented in [165] eliminates FLOPs, multiplications, and nonlinear operations during inference. All multiplications are precomputed and stored in a multiplication table. Thus, the FLOPs are omitted by table lookups and fixed-point summation, and nonlinear operations are omitted by quantizing the activation functions. An alphabet set multiplier (ASM) that uses computation sharing is used in [166] to reduce the cost of multiplication in DNN. In ASM, conventional multiplication is performed using a precompute, adder, shifter, and select unit. The precompute bank finds the alphabet using the product of input and some smaller bit sequences. Shifters and adders are then used to obtain the multiplication between an input and an alphabet. The result of this computation is shared. Therefore, based on the weights, the complete multiplication can be realized through a proper combination of the alphabet using the shifter and adder.

E. Computational Optimization

Approximate computing (AC), low-rank approximation, and stochastic computing (SC) are three major computational optimizations used to compress the DL models targeting edge devices. At acceptable accuracy degradation, hardware realization of DL models with reduced complexity, latency, and improved energy efficiency is achievable. AC can be applied at both the algorithmic and hardware levels. For example, at the algorithmic level, loop perforation [70] skips some iteration by trading off between accuracy and latency or power consumption. At the hardware level, an approximation could be in architectural optimization [167], approximate data storage [168], and reading writing speed modifications [169]. In [170], the approximation of both computation and memory access for NN demonstrates 34%–51% energy savings within 5% quality degradation. Based on a ranking of the contribution of neurons to energy consumption and accuracy, less important neurons can be approximated. AxDNN [171] is the cross-layer approximation of a DNN incorporating activation pruning, approximate multiplier, and voltage scaling to improve energy efficiency and speed. An approximated DBN is presented in [172]. AxTrain [173] also leverages AC that improves the inference on hardware by finding the error-tolerant parameters. An approximate NN (AxNN) can be obtained using a quality-configurable technique exploiting neuromorphic processing [174].

The low-rank approximation is another effective way of compressing the size of the DL model. In DL, there is a significant redundancy at filters/channels to be encoded using low-rank matrices to get an approximate network. For instance, redundant filters of an NN layer can be compressed using the linear combination of fewer filters. A data-driven approach is presented in [175] that uses Tucker decomposition to obtain a low-rank approximation for various image classification models demonstrating

improved computation and reduced bandwidth requirements. Tensor decomposition can be applied to a trained DL model to decompose the large filters into several smaller tensors. It can improve performance without significant degradation of accuracy. In canonical polyadic (CP), decomposition can be combined with low-rank approximation to increase the model compression. A 4.5× speedup of CNN in CPU has been achieved in [176]. Fine-tuning of weights can restore the loss of accuracy caused by CP decomposition at certain levels. One of the advantages of low-rank approximation in DL is the simplified model architecture with reduced parameter count.

SC allows MAC operations through AND gates and multiplexers that help to reduce power consumption. SC has shown promise in realizing the DL inference on embedded systems utilizing simplified arithmetic operations. An efficient implementation of a DCNN is presented in [177] based on SC for activation functions, pooling, and convolution layers. SC can also be used in ReLU, parallel counter, and near-max pooling operations' approximation to optimize the CNN [124]. There are significant efforts to reduce the computation time and latency of SC-based multiplication in DL using logarithmic quantization [179] and differential MAC [180].

F. Knowledge Distillation

KD is a compression method that mimics the behavior of a large DL model into a lightweight model by transferring learned knowledge [178]. The conceptual representation of the KD technique is demonstrated in Fig. 11. Two different models are used for training and inference. Usually, the large network is called the teacher model, whereas the small one is called the student model. The student models are the compressed model having a smaller size that can be deployed in resource-constrained devices to make the efficient inference. The transfer of knowledge is done by training the student model to imitate the teacher network through the minimization of a loss function. The idea of KD was developed in [181] and generalized in [182]. In [182], the student model was trained to imitate the class probabilities of the softmax output of the teacher model. An improvement of 2% accuracy for speech recognition has been demonstrated in using the KD methods. The student network achieves similar accuracy as the teacher network consisting of an ensemble of ten networks. The student model can also use other information from the teacher model, such as activations [183], neurons [184], or features of intermediate layers [185] as knowledge to improve the performance.

In KD, the teacher model can be significantly larger and pretrained. There are research efforts to design an effective student model. For example, DeepRebirth [186] merges convolution layers with consecutive weightless layers, such as pooling and normalization, together to get a compressed student model. The parameters are then fine-tuned layer-wise to minimize the accuracy degradation. DeepRebirth has shown promising results by 3× speedup and 2.5× memory reduction. Rocket launching [187] is a parallel learning scheme that trains both the teacher and student models simultaneously.

For NLP on edge devices, a smaller transformer-based model has been designed using the KD technique named TinyBERT [188]. The knowledge encoded in the teacher BERT is effectively transferred to the TinyBERT maintaining at least 96.8% performance of larger BERT model with 7.5× reduced model size and 9.4× inference speedup. Another technique is presented in [189] based on the extension of KD for sentence transformer distillation demonstrating compressed student models with comparable performance.

FitNets [185] are student networks with increased depth and reduced width that make a generalized compact network. A guided layer introduced in the middle of the student layer learns from the hint layer in the teacher network. Result demonstrates comparable classification accuracy for the CIFAR-10 dataset with speedup and reduced parameter count. Various applications with impressive results have been developed for KD, such as mutual learning [190], teacher assistant [191], and self-learning [192]. Also, the idea of knowledge transfer has been extended to dataset distillation to obtain a compressed training dataset reducing the computational burden of training a DL model with a large amount of data. The knowledge from a large dataset is transferred to a smaller dataset that significantly reduces computational loads of training DL models [193], [194]. KD is a very flexible compression technique that can be adapted for a wide variety of applications. The major drawback of KD is the unbounded decision boundary of student and teacher models.

G. Adaptive Optimization Techniques

Most DL compression techniques, such as pruning, quantization, and KD, are static. The optimizations are independent of the change in input. Sometimes, the decision-making complexity of DL algorithms depends on the difficulties of input. For example, object detection from a blurry image is more difficult than detection from a clear picture. Similarly, for NLP, recognizing a simple sentence is much easier than a complex sentence. Adaptive optimization techniques allow tuning during inference runtime based on the current input to the network. To avoid redundant computation of video frames, NoScope [209] skips frames having insignificant changes to adjacent frames. A difference detector is used to identify the changes between consecutive frames employing a lightweight classifier.

Two DL models of different sizes (big and little) can be used for classification during inference [210]. The little network runs first assuming that easy input occurs more frequently than difficult ones. A success checker evaluates the classification confidence. The inference is forwarded to the output when the confidence score exceeds a predefined threshold. Otherwise, the big model is employed to

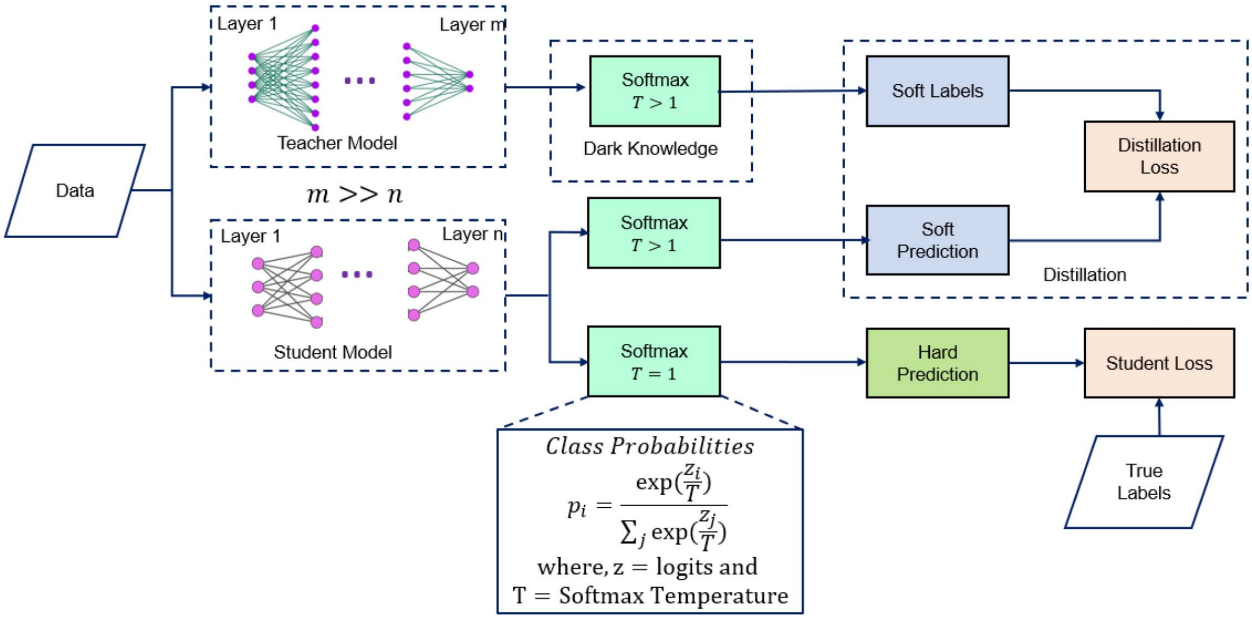


Fig. 11. *Illustration of KD [178]. The knowledge from the pretrained large teacher model is transferred to the shallow student model through the minimization of distillation loss.*

execute the inference. This technique offers up to 53.7% and 94.1% energy savings for image classification and digits' recognition, respectively. The little/big scheme will save energy only if the big model is seldom activated. It also requires double training time and resources for two networks. The learned parameters from the two models consume additional on-chip memory.

Instead of storing weights from two separate models, incremental training and weight reuse can improve some limitations [211]. With increased depth, the DL model learns from simple to complex features. Therefore, if the input is easy, a shallower network may make accurate decisions, whereas deeper models classify complex inputs as having very complicated nonlinear decision boundaries. Instead of running through the last layer of the network, part of the network can be turned off during the inference from easy input.

The early exiting technique can reduce the latency by generating acceptable inference performance using only a portion of the complete DNN model. An early exiting CNN architecture with four exit points is demonstrated in Fig. 12. BranchyNet [212] is an implementation of an early exiting technique that adds multiple exit points to the base network. The confidence score is calculated as the entropy of softmax. Comparing the confidence score with a threshold, the inference is either stopped or continued. Instead of making an inference at the end of the network, an inference result can be produced at any of the exit points improving latency and saving resource utilization. However, improved performance is only achievable if the deeper branches are rarely used. In [213], a regularization has been added to the latency term to identify whether the current input sample is to be passed to the next layer or not.

Early exiting concepts can be extended to different computation levels. For instance, DDNNs [214] are based on BranchyNet having exit points at three different network levels: cloud, edge server, and end-device. This concept is also used in DeepIns [215] for inspecting manufacturing processes in the industry. In DeepIns, data are collected by the edge sensors and have two exit points: one at the edge server and the other one at the cloud. An authentic operation (AO) unit is added to BranchyNet in [216] to determine whether the input is to be passed to the edge server or cloud based on the confidence score.

Due to additional weights from the lateral branches, the BranchyNet requires more storage than the original network. Instead of adding branches, skippable tiny NNs known as gates are attached to the network architecture of SkipNet [217]. The concept of SkipNet is illustrated

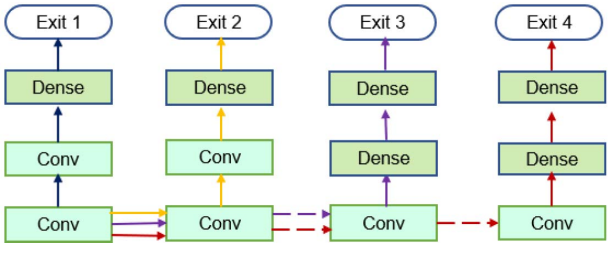


Fig. 12. *Example of early exiting technique [18]. There are four exit points in the CNN classifier having different depths.*

Table 3 Summary of Model Compression Techniques and Results of Compression

Ref.	Goal	Base Model	Compression Technique	Performance Type	Results
[121]	Reduce model size	Faster R-CNN	Quantization	improved	$4.16x$ smaller
[140]	Reduce model size, speedup	AlexNet	Binarization	lossy	32x reduced size, $58x$ faster
[157]	Reduce model size	VGG-16	Pruning, Quantization, Huffman coding	lossless	$49x$ size reduction
[195]	Reduce computation	VGG	Pruning	lossless	$5x$ reduced computation
[108]	Improve energy efficiency	AlexNet	Pruning	lossy	$3.7x$ energy reduction
[105]	Faster inference	VGG-16	Pruning	lossy	$10x$ faster
[196]	Speedup	ResNet	Pruning	improved	$20x$ faster
[47]	Reduce storage	AlexNet	Pruning	lossless	$13x$ reduction
[197]	Speed up, reduce model size	NIN	Pruning	lossy	$1.24x$ faster
[198]	Resource utilization, computation optimization	SqueezeNet	Compact network design	improved	$5.17x$ smaller
[73]	Reduce model size	AlexNet	Compact network design	lossless	$50x$ reduction
[199]	Reduce model size, speedup	YOLOv2	Compact network design	improved	$15.1x$ smaller, 34% faster
[200]	Reduce storage, speedup	AlexNet, VGG	Kernel separation, low-rank approximation	lossless	$11.3x$ less memory, 13.3x faster
[175]	Improve energy efficiency	VGG-S	Low-rank approximation	lossy	$4.26x$ reduced energy cost
[201]	Reduce computation	VGG 16	Low-rank approximation	lossless	9x faster
[202]	Reduce Computation	CNN	Low-rank approximation	lossy	$2x$ faster
[203]	Speedup	CNN	Low-rank approximation	lossy	4.5x faster
[186]	Reduce storage, faster	GooLeNet	Knowledge distillation	lossy	$2.5x$ reduced memory, 3x faster
[204]	Reduce storage	WideResNet	Knowledge distillation	lossless	40% reduction
[185]	Generate lite network	FitNet	Knowledge distillation	improved	Increased accuracy, reduced model size
[205]	Generate lite network, improve performance	ResNet	Knowledge Distillation	improved	1.1% increased accuracy
[206]	Minimize storage	NIN	Knowledge distillation, regularization	improved	$33.28x$ smaller memory
[207]	Speed up, energy saving	DNN	Dataflow optimization	improved	$58x$ faster, 104x energy saving
[208]	Power, performance tradeoff	VGG	Memory access optimization	lossy	83% accuracy
[65]	Speed up	MobileNet	Adaptive optimization	lossless	$1.7x$ speedup

in Fig. 13. In the inference runtime, the network can skip some layers depending on the input type to save computations with negligible effect on accuracy. Splitting the inference into multiple tasks and sequentially executing with increasing complexity saves energy. For example, speech recognition in smart assistants (e.g., Google Home, Apple Siri, and Apple Watch) uses two networks. A small RNN is used on the edge that can run on energy constraint wearables, such as smartwatches with low power consumption [19]. After the wake-word detection, the rest of the audio signal is transmitted to interpret and take actions in a large speech recognition DL model running on the cloud. Such a hierarchical face recognition method using a smartphone is presented in [218], which uses CNN. Steps of CNN are executed in the custom inference engine sequentially with increased complexity. A medical diagnostic tool based on hierarchical inference is developed using wearable sensors in [219]. First, a small CNN on the edge device is used to predict whether the person is ill or healthy. If the patient is ill, then a deeper and larger model is used on the cloud that uses more complicated diagnoses using multiple sensor data.

Representative model compression techniques from this section are summarized in Table 3. This table presents the summary from multiple viewpoints, such as objectives of the corresponding research, base network, and adopted compression techniques, whether the outcome is lossy, lossless, or improved, and finally, results are tabulated.

Table 4 presents a summary of popular lightweight DL models with the number of parameters and operations. Most of the models listed in Table 4 are the results of applying compression techniques on the computation-intensive DL models.

VIII. HANDLING SPARSE TENSORS FOR EFFECTIVE EDGE INFERENCE

The DL models sparsity can be of two types: structured and unstructured. The nonzero elements are randomly scattered in unstructured sparsity and follow a regular pattern in structured sparsity. Some of the basic operations,

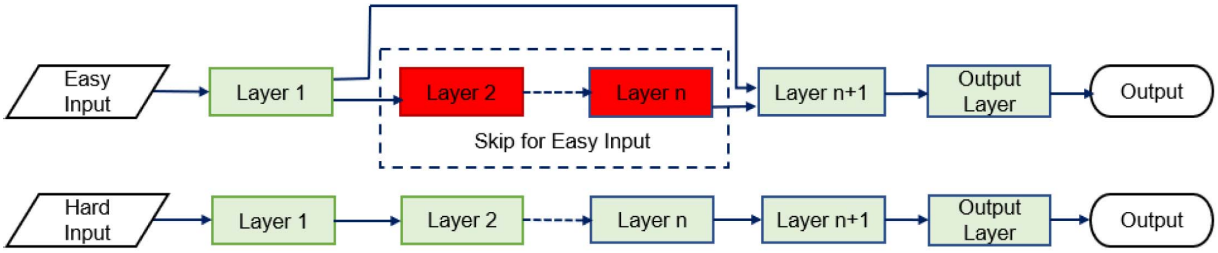


Fig. 13. *Example of skipping net [217]. The network can skip some layers depending on the input type to save computations with negligible effect on accuracy.*

such as dropout, quantization, and ReLU activation, result in unstructured sparse activations and weights that are inherent in the DL models.

Activation functions, such as ReLU, clamp the negative inputs to zeros resulting in sparse tensors. For instance, activations are 40% and 70% sparse in CNN and FC layers, respectively, [220] due to ReLU activations. Max pooling operation can boost the sparsity up to 80% for the VGG-16 network [100]. Upsampling operation in CNN inserts zero weights resulting in sparsity [221]. Transposed convolutions in the GANs insert zeros by upscaling resulting in 60% MAC elimination [222]. Dropout is a technique often used in DL training to prevent overfitting that introduces activations sparsity [48].

The pruning technique can remove above 90% and 60% weights of dense layers and convolution layers, respectively, introducing significant weight sparsity [47]. For compact network architectures, such as MobileNet-v2 and EfficientNet-B0, pruning can introduce 50%–93% sparsity resulting in 2.5×–4.2× MAC elimination [223]. NLP models, such as Transformer [45] and BERT [224], have around 80% [225] and 93% [226] weight sparsity. If exploited properly, such sparsity can result in 4.8× and 12.3× MAC reduction for Transformer and BERT, respectively. Lossless reduction of computation and memory access of 3.8× and 1.1× is reported in the NLP model SpAtten [227] by pruning unimportant tokens and heads.

Above 80% weights can be pruned from RNN [228], GRU [160], and LSTM [229] models without any significant loss of accuracy. For instance, refactoring the batch normalization for RNN in MASR [230] achieves about 60% activation sparsity. Through activation sparsification, DasNet [231] reports MAC elimination of 27% for AlexNet and 12% for MobileNet with less than 1% top accuracy degradation.

The advantage of such sparsification has been examined in CNN models to be about 40% and 50% MAC operation elimination during training and inference, respectively [232]. In MobileNet-v2, activation sparsity can eliminate 20% MAC operation [72]. The execution time and energy are significantly reduced by exploiting sparsity by eliminating ineffective calculations and considering only nonzero computations. Significant speedups are possible by reducing memory access and storage requirements by storing only the nonzero values.

Hardware accelerators for efficient inference may exploit the sparsity of weights, activations, or both. Accelerators can handle the sparsity both in a static way or dynamically. An example of static sparsity involves the locations of zero weight being known beforehand for inference in Cambricon-X [233]. Such architectures allow off-line structured arrangements of sparse tensors [234]. Recent developments of inference accelerators, such as ZENA [235], SNAP [236], and EyerissV2 [237], leverage dynamic sparsity. Dynamic sparsity handling accelerators are required to determine the locations of nonzero elements in tensors.

The sparsity in the DNN accelerator can be exploited at two levels: neurons and weights/activations. The MAC operations can be skipped for zero weights as implemented in Cambricon-X [233]. The sparse weights are stored with the address index. The activations are fetched to the PEs according to the address index of the weights, and the MAC operation is performed. Thus, exploiting the sparsity of weights, the number of MAC can be reduced, which, in turn, reduces the latency and energy consumption. However, the distribution of nonzero weights is irregular causing address indexing problems, inefficient memory access, and imbalanced PE. The number of address indexes is reduced in the block sparsity, where a single address index is used for a group of weights [238], [239]. Architectures exploiting block sparsity require less MAC than

Table 4 Lightweight DL Models for Edge

Lightweight Model	Number of weights (M)	Number of operations (G)
SqueezeNet [73]	1.25	1.72
SqueezeNext [74]	3.2	1.42
MobileNet-v1 [70]	4.2	1.15
MobileNet-v2 [72]	3.4	0.60
MobileNet-v3-Small [64]	2.9	0.11
CondenseNet [76]	2.9	0.55
NASNet-B [59]	5.3	0.98
PNASNet [66]	5.1	1.18
MNASNet-small [67]	2.0	0.14
ANTNets [77]	3.7	0.64

weight-sparse architecture, and the PE is more balanced with efficient memory access. Vectorwise block sparsity is exploited in [239], where nonzero weights from adjacent multiple columns are combined with a column index. A multiplexer is used to correctly identify the column index and the corresponding input.

The neuron sparsity in the input and output is exploited in many DNN accelerators. The number of operations can be lessened by observing the sign of accumulated activation in the PE [240]. In [240], the negative values are detected to terminate the computation because negative activation will be zeroed by ReLU and will remain zero in subsequent layers. The computations for only the important neurons are realized in [241]. The predictions are performed first, and then, the output indexes are used to exploit the sparsity. The bit-serial processing for MAC calculation is used to employ the flexible bit width. Architectures can also exploit both the weight and neuron sparsity in a joint way [242], [243], [244].

Sparse weight compression is an effective technique that results in 20%–30% memory access reduction [157]. The compressed sparse weight matrix can be saved in two main formats: compressed sparse row (CSR) and compressed sparse column (CSC). Eyeriss v2 [237] uses the CSC format to store compressed parameters and activations. It provides support for both sparse and dense models. A hierarchical mesh ensures high flexibility in PE interconnections. Cnvlutin [245] uses the CSR scheme for compressed activation without exploiting the weight sparsity. In Cambricon-X [233], the PEs used compressed weights for calculation but do not consider the activation sparsity. The SCNN [244] supports compressed processing for convolutional layers, and EIE [243] supports processing for dense layers. In SCNN [244], the CSC scheme is used for storing both weights and activations. A multiplier array generates the products from compressed values that are summed using interconnection mesh. Sparten's [246] architecture is similar to SCNN with an improved multiplication reducing the overhead. EIE [243] performs sparse multiplication for dense layers. Weights are stored using the CSC format and have the zero-skipping ability for null activations. EIE also stores the starting position of each column, which is required due to the variable length of compressed weights. For nonzero input, the compressed weight column is read to update the output. Moreover, power savings are achieved by avoiding external DRAM access. The CSC provides better memory efficiency compared to the CSR format [247]. Compression methods for sparse-weight matrices also include compressed image size (CIS) and run-length coding (RLC). NullHop [248] utilizes the CIS scheme for the weights and implements zero skipping for null activations. SqueezeFlow [249] used the RLC for weights supporting both the sparse and dense models. RLC for activation compression, PE data gating, and zero input detection are used in [250]. The concise convolution is introduced to avoid null results. Unique weight CNN (UCNN) accelerator [251] reduces the memory size considering a generalization of the sparsity. Instead of utilizing the repetition of zero weights, UCNN considers the repetitions of weights having any value and reuses the computations.

IX. QUALITY METRICS FOR EDGE INFERENCE PERFORMANCE EVALUATION

Comparisons between different edge hardware platforms for DL inference are not straightforward. There are some standard metrics to define the performance of the DL inference on edge devices described in this section.

A. Model Size

The size of the model is an important metric to evaluate EI performance. Compact network architectures are preferred for edge devices. Hyperparameters such as the number of layers, nodes per layer, activation functions, the complexity of computation per layer, loss functions, and the number of connections between nodes of one layer to the next significantly affect the inference performance. Therefore, the network architecture and hyperparameters of the model need to be considered. In addition, the number of parameters (weights and biases) reflects the memory requirements and, therefore, acts as another important metric.

B. Accuracy and Robustness

The accuracy verifies if the inference engine can accomplish the assigned task properly. Some EI applications, such as autonomous driving and face verification for device unlocking, require very high accuracy. Inference accuracy, sometimes, depends on the input feeding speed. For instance, while processing video frames, a few frames may be skipped due to device constraints and fast feeding rate resulting in an accuracy drop. Robustness also plays a significant role in evaluating inference performance. In many cases, a highly accurate model with ideal datasets and a simulation environment will produce poor results in a practical scenario. As inference on edge is focused to bring DL solutions to the end user, robustness is important. The robustness of the accuracy can be determined by evaluating the performance on widely accepted multiple datasets. Also, if any data preprocessing improved the accuracy, that should be carefully considered during inference.

C. Area and Cost

The chip area represents the total silicon area required to implement all the computations for the inference operations. The area is often expressed in squared millimeters (mm^2) and squared micrometers (μm^2) units. It is dependent on the technological node, architecture, the number of cores, and the size of the on-chip memory. The expense of the chip depends on the chip area requirement, including the size and type of storage, and the amount of control

logic. The typical on-chip memory capacity of inference engines is in the range of a few hundred kilobytes. Thus, in addition to chip area, the required memory sizes should be carefully considered.

D. Power Consumption

The power consumption often expressed as milliwatt (mW) or microwatt (μW) is the total power required to execute the inference. The power consumption report should include the power consumption of memory access. Although alternate power sources are emerging for edge devices through energy harvesting [252], reducing the power consumption of inference is crucial. The number of computations and off-chip memory access implicitly impact the power consumption. Thus, the amount of off-chip memory accesses should be carefully considered. Power consumption caused by memory access can be predicted by the off-chip data read and write per inference.

E. Energy Efficiency

It measures the rate of operations that a computing device can deliver for every watt of power consumption. Usually, the edge devices are battery-operated, requiring ultralow power consumption and highly energy-efficient operations. Energy efficiency is one of the key criteria to evaluate the performance of DL inference. It is often expressed as giga operations per watt (GOPS/W) or tera operations per watt (TOPS/W), illustrating how many operations the inference engine can perform per watt of power consumption. The FLOPs per second (FLOPS) per watt (FLOPS/W) metric is also used to assess the performance of a computing system that involves many floating-point calculations. The energy efficiency is influenced by the DL model size, the number of memory access, and computation needs.

F. Latency

Latency defines how fast an inference engine can accomplish a complete inference. For real-time performance, improving the latency is crucial. Latency is often expressed in units of time (microseconds, milliseconds, and seconds) that measures the difference in time between the arrival of the input signal to the time when the result is generated. Low latency is the desirable feature of EI. For instance, real-time augmented reality (AR)/virtual reality (VR) applications and intelligent robotic vision may put stringent latency requirements to a few milliseconds. Latency depends on many factors, including the types of edge devices, available resources, and how the inference operations are being processed.

G. Throughput

The throughput of DL is defined as the maximum number of inputs that the network can process per unit of time. It is often expressed as the number of operations that an inference accelerator can perform per second. Higher throughput indicates better performance. Metrics used to quantify the throughputs are giga operations per second (GOPS) and tera operations per second (TOPS). As the major computations of DL algorithms are MACs, therefore, throughput is sometimes expressed as giga MACs per second (GMAC/s) and tera MACs per second (TMAC/s). An MAC consists of multiplication and addition (two operations); thus, the ratio between TOPS and TMACS is 2:1. High throughput is imperative to provide real-time execution for numerous use cases in navigation, medical diagnosis, security, and automation. The low latency and high throughput are expected to run fast inference for real-time performance. The throughput is related to the number of cores in the chip. Therefore, the number of cores is also an important parameter to be considered.

H. Memory

Optimizing the memory requirement plays a vital role in high energy efficiency and low power consumption EI. A highly accurate DL model requires computations of millions/billions of parameters that consume large storage. On edge devices, there is an obvious scarcity of memory. Thus, the memory footprint is an important quality metric to evaluate the inference performance. The memory requirements are mainly contingent upon the DL model size and required number of memory access, and the types of memory used. Lowering the memory requirement is the better optimization of the DL inference on edge.

I. Test Setup

The test setup is an important aspect of evaluating the DL inference for different applications. For example, one processor can be designed to report very low power consumption without considering off-chip memory access. When evaluating the overall system performance, the off-chip memory would cause substantial power consumption. Similarly, it is important whether the results are produced by simulation or experiments. With much preprocessing in ideal simulation cases, DL can generate superior performance, which can deteriorate significantly in real-life experimental case.

X. SOFTWARE TOOLS AND TECHNIQUES FOR EDGE INFERENCE AND CODESIGN

The DNN inference on the edge is slower compared to the cloud. Thus, to accelerate the performance, both hardware and software optimizations are carried out. Hardware acceleration mainly focuses on parallel computing, while software optimization focuses on pipelining, resource management, and efficient compiler design. The low-level libraries are mainly developed for GPU computation and are not widely available for edge devices. However, these libraries (e.g., cuDNN, CUDA, and MKL-DNN) provide an optimized implementation of the building blocks of

DL methods, such as forward and backward propagation, activations, pooling, convolutions, and normalization. For inference on the edge and IoT devices, hardware-specific development platforms, such as Nvidia Jetson TX2 [253] and Intel Edison kit [254], are available.

Open-source frameworks are highly successful for DL model building, training, validation, and evaluation. TensorFlow [255], Caffe [256], and PyTorch [257] are some of the popular frameworks. TensorFlow by Google uses the directed graph for computation and a placement algorithm to distribute the computation task on nodes. TensorFlow enables posttraining quantization to 8 bits using a scaling factor with a bias to map the weights within the INT8 range. TensorFlow Lite (TF-Lite) and TensorFlow Micro are two optimized extensions of TensorFlow for mobile, embedded devices, and microcontrollers, respectively, to provide on-device inference from pretrained models [258]. TF-Lite and Caffe can efficiently load and compute sparse matrices. These frameworks facilitate a significant reduction of latency, memory, the number of FLOPs, power, and inference time in pruned models than their dense counterparts. RSTensorFlow [259] is another extension of TensorFlow, developed to ensure proper utilization of the heterogeneous computing resources to run inference on android devices. It has demonstrated improved matrix multiplication and speedup of different DL models on Android devices with GPU support. Caffe and PyTorch are two other frameworks maintained by Facebook. Caffe focuses on DL model building and experimentation for edge devices, such as smartphones and Raspberry Pi. PyTorch is a very convenient platform to deploy DNN research prototypes into production. Open NN Exchange (ONNX) [260] enables interoperability, hardware optimization, and portability within different frameworks. A model trained in one of the frameworks can be deployed to make inferences using another framework taking advantage of the ONNX standard format.

Graph compilers take the computation graph to generate optimized instructions for specific hardware. Graph compilers (e.g., Tensor-RT, TVM, XLA, and OpenVINO) often optimize the graph structure by merging redundant operations, performing kernel autotuning, enhancing memory reuse, preventing cache misses, and so on. Table 5 presents an outline of the widely used software packages for algorithm–hardware codesign and optimization. This table also includes the supported ML frameworks and edge devices by each of the software packages.

A. Compression-Compiler Codesign

The compression-compiler codesign has recently gained attention demonstrating improvement in EI performance in mainstream general-purpose edge devices. The DL model compression and compilation of executable codes are performed in synergy to ensure optimum model size and inference speed. The compilation is the mapping of high-level DL operations into hardware-level instructions. This codesign approach remained unexplored for a long and has recently shown promise in improving energy efficiency and latency. A few prominent examples include PCONV [261], PatDNN [262], and CoCoPIE [263], which considers simultaneous design exploration compression and compiler. In model compression, structured pruning obtains compiler-compatible patterns. In the compiler design phase, parallelism at both the instruction and thread levels generates efficient compiler code. MCUNet [264] is a framework incorporating compact model design and compiler optimization to eliminate instruction and memory redundancy. Compression-Compilation codesign for Performance, Intelligence, and Efficiency (CoCoPIE) [263] is a software framework that effectively uses the pruning and quantization techniques compatible with compiler instructions. The two-part optimization in CoCoPIE includes CoCo-Gen and CoCo-Tune. The CoCo-Gen is the pattern-based structured weight pruning and pattern-aware executable code generation. A simplified overview of the CoCo-Gen framework is illustrated in Fig. 14. The synergy in fine-grained pruning for high accuracy and hardware-compatible compiler code generation resulted in the efficient execution of DL inference. However, the process is slow due to the time-consuming pruning process. The CoCo-Tune fastens the pruning by effectively identifying the parameters to discard through a composability-based compiler framework. In CoCo-Tune, the pruning is speeded by considering the CNN model as a composition of pretrained blocks and reusing them to identify the components to be discarded. In addition, computation reuse is exploited by utilizing hierarchical compression techniques. An experimental demonstration of CoCoPIE resulted in improved energy efficiency and inference latency for ResNet50 and VGG-16 models in general-purpose smartphones compared to neural accelerator ASICs and FPGAs [274]. This technique has also been experimented with for object detection using YOLO-v4 and 3-D activity detection using C3D, R(2 + 1)D, and S3D networks achieving 19 frames/s and 6.8 ms per frame performance without accuracy degradation, respectively, in smartphones [275]. Therefore, the compression-compiler codesign has emerged as a promising technology to ensure energy efficiency and faster DL inference in general-purpose edge devices.

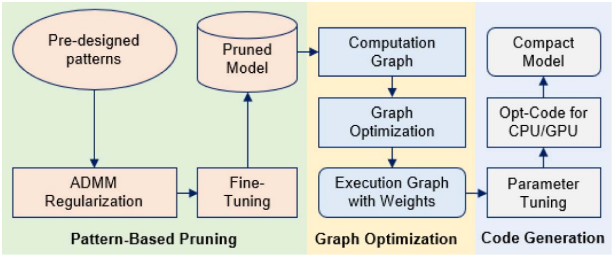


Fig. 14. *Overview of compression-compiler codesign used in CoCo-Gen of the CoCoPIE software framework [274].*

Table 5 Software Packages and Supported Edge Devices for Algorithm–Hardware Codesign

Software for DL inference on edge	Supported frameworks	Supported edge devices
Intel OpenVINO Toolkit [258]	Caffe, TensorFlow, ONNX	Intel® CPU, Integrated Graphics, Neural Compute Stick 2, Movidius™ VPUs, and FPGAs
Matlab Deep Learning HDL Toolbox [265]	Keras, TensorFlow	Xilinx Zynq®-7000 ZC706, UltraScale+™ MPSoC ZCU102, Intel Arria® 10 SoC
XCUBE-AI [266]	Keras, TensorFlow Lite, ONNX standard format	STM32 Arm® Cortex®-M-based MCU
XILINX DNNDK [267]	Caffe and TensorFlow	Xilinx® Zynq®-7000 and Zynq UltraScale+™ MPSoC
NVIDIA TensorRT [268]	TensorFlow, MATLAB, ONNX	Tesla P4, Tesla V100, Drive PX2, Jetson TX2, NVIDIA DLA
CEVA Deep Neural Network (CDNN) [269]	Caffe, TensorFlow, ONNX	CEVA-XM Vision Processor, NeuPro, and SensPro
Qualcomm® Neural Processing SDK [270]	Caffe/Caffe2, TensorFlow ONNX	Qualcomm® Snapdragon mobile chips (Hexagon™ DSPs, Adreno™ GPUs, Kryo™ CPUs)
Cadence Stratus HLS [271]	TensorFlow, Caffe	RTL/FPGA
TensorFlow Lite [272]	TensorFlow	Android and iOS mobile phones, MCU (ARM Cortex-M, ESP32)
Embedded Learning Library (ELL) [273]	Microsoft CNTK, Darknet, ONNX	Raspberry Pi, Arduino, micro: bit

B. Algorithm–Hardware Codesign

The processing of DNN algorithms in hardware for efficient inference is attracting continuous research attention. The reason for the current surge of customized hardware design is twofold: 1) conventional hardware performance is reaching the upper limit and 2) adopting the compression techniques in hardware is a potential choice to meet the increasing computation demand. The paradigm in customized DNN accelerator design is the algorithm–hardware codesign. A general framework of the algorithm–hardware codesign is illustrated in Fig. 15. The algorithmic design can be performed in any of the ML frameworks. The best-learned models are then used by an edge-specific intermediate software package for optimization. The optimized models are used for inference acceleration on the target hardware platform. The algorithmic optimization provides effective hardware design instructions. On the other hand, the hardware realization of DNN algorithms provides feedback for more efficient algorithm design. Therefore, the algorithm–hardware codesign is the new research direction in obtaining customized application-specific DNN accelerators.

C. Frameworks to Exploit Hardware Parallelism

The goal of parallelization is to maximize performance through proper utilization of the available hardware

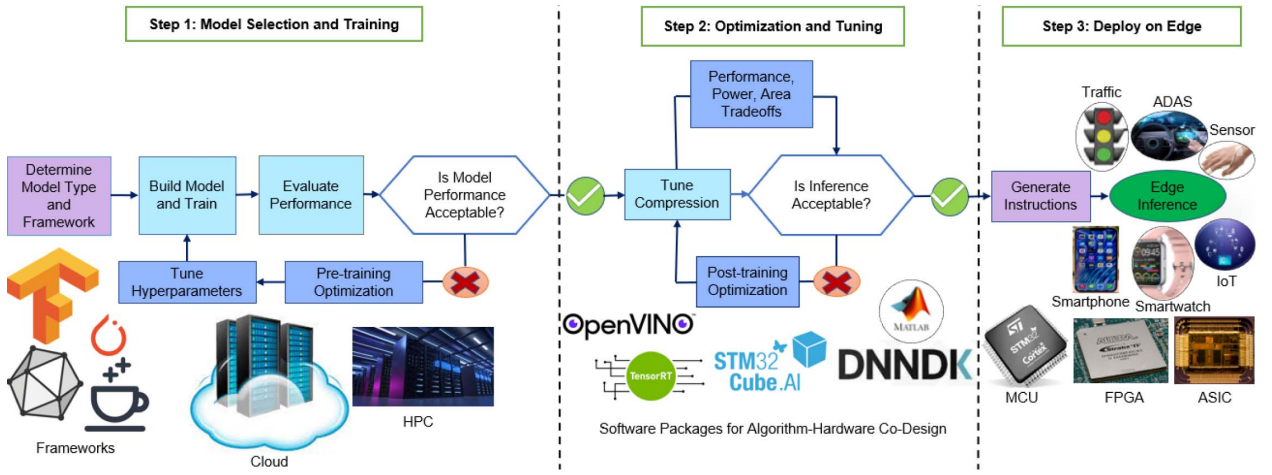


Fig. 15. ***Block diagram of the generic training to inference pipeline. The algorithmic design can be performed in any of the ML frameworks. The learned model is then used by an edge-specific intermediate software package for optimization. The optimized models are used for inference acceleration on the target edge hardware platform.***

resources. Parallel computing of CNNs and RNNs has demonstrated faster response in CPU, GPU, and DSP [259]. For instance, CNNdroid [276] is a GPU-based library for DCNN inference on the android smartphone that can result in 60× speedup and 130× energy reduction.

Nvidia's compute unified device architecture (CUDA) is a parallelization framework that can be used for embedded and mobile devices [277]. For instance, vectorized implementation of convolution can be used in real-time object detection and robotic controllers. However, directly employing the CUDA for mobile GPU may deteriorate the system performance [278]. RenderScript [279] is another parallelization framework for android devices to automatically exploit data structure parallelization among accessible GPU cores. SqueezeNet implementation using RenderScript has demonstrated a 310.74× speedup on Nexus-5 mobile devices. The Cappuccino [280] framework can automatically synthesize the appropriate DL models for EI. Cappuccino takes in the model definition, model file, and dataset as input and applies three levels of parallelization on kernel, filter, and outputs. Using cappuccino, faster response and high energy efficiency can be achieved at the expense of imprecise computation.

DeepSense [281] is a GPU-based CNN inference framework for mobile devices with improved latency. In DeepSense, most computation-intensive tasks are performed by GPU, and only the padding and memory allocation are performed by the CPU. Limited memory of mobile GPU negatively impacts running the DCNN on mobile devices [282]. Thus, splitting the computation of convolution and dense layer between GPU and CPU can achieve a 60× speedup for object detection using YOLO in the Jetson TK1 board [283].

DeepX [284] uses SVD to compress layers to get compact network architecture. Then, the architecture is decomposed into blocks to run parallel inference. The DeepX toolkit (DXTK) [285] is a generalization of DeepX techniques for other DL models and layers.

D. Efficient Memory Access

The basic operation of DL is the billions of MAC. For the successful computation of each MAC, a total of four memory accesses are needed. The memory accesses are three read operations to fetch the weight, activation, accumulated partial sums (psums), and one memory write to save the updated sum. Therefore, one of the challenges in DL inference is efficient memory access. The generic architecture of a DNN inference accelerator is presented in Fig. 16. In general, the DL inference accelerating hardware follows a hierarchical memory structure for efficient processing. The outer layer of the memory hierarchy is the off-chip DRAM that stores the weights and activations. There are three SRAM buffers or global buffers (GLBs). The GLB stores IAs, weights, and output activations in the intermediate layer to feed the PEs. The lowest level of the memory hierarchy is the register files (RFs) within the PEs. In the worst case scenario, if all four off-chip DRAM accesses are required for MAC calculation, the throughput and energy efficiency are significantly affected.

DRAM access is the most power-consuming task in the memory hierarchy. DRAM in the DL accelerator can store more data requiring more energy compared to the on-chip small memory. The on-chip memory is usually a few kilobytes in size. Instead of frequent access to DRAM, storing the available data on-chip to reuse is an energy-efficient solution. A high number of DRAM access results in increased latency and energy costs. Thus, many techniques have been introduced to reduce the number of DRAM accesses. In ShiDianNao [286], the inference architecture is embedded inside the camera sensor. Thus, eliminating any DRAM access to store the data leads to 60× energy savings compared to DianNao [287].

Thus, reusing data stored in faster and low-energy small memories, such as GLBs and RFs, is preferred. However, due to the limited storage capacity of on-chip memory, different dataflow techniques are developed. In CNN, three types of data can be reused, such as inputs, weights, and convolutions. The inputs captured in the adjacent receptive fields can be reused to calculate more than one output value. Weights are often redundant for neighboring filters and can be reused. When the weight slides through the input to compute the convolution, overlapping regions can be reused. During the inference engine design, efficient data flow can improve the throughput.

There are four types of dataflow schemes used for DL acceleration [2]: 1) weight stationary (WS); 2) no local reuse (NLR); 3) output stationary (OS); and 4) row stationary (RS). In WS methods, the weights read from the DRAM are stored in the RFs and remain stationary. The PEs reuse the same weights for as many MACs as possible, therefore minimizing the power consumption of memory access for reading weights. For example, neuFlow [288] is a WS dataflow processor for computer vision applications. In NLR, weights are not stationary inside the PEs. In DianNao [287], IAs and filter weights fetched from the buffer are used in the MAC unit to get the psums. The sums are then stored in registers to be reused by the PEs. Thus, reducing the memory access for psum saves energy. ShiDianNao [286] uses the OS dataflow scheme. The energy efficiency is improved by reusing the activations from neighbor PEs. The RS dataflow [250] considers the efficient data reuse at the lower level of memory hierarchy like RFs. The inputs, weights, and psums are reused to update the psum through accumulation at the registers to increase energy efficiency. The PEs of MAERI [289] consist of RFs and multipliers. MAERI allows flexible dataflow through reconfigurable interconnects, distribution trees, and adder trees. The distribution trees are used to supply the activations and weights to the PEs, and the adder trees are used to collect the multiplier outputs. Sparsification techniques can be exploited using compression to reduce memory access and memory size. For example, external memory bandwidth can be reduced by 1.5× for activations

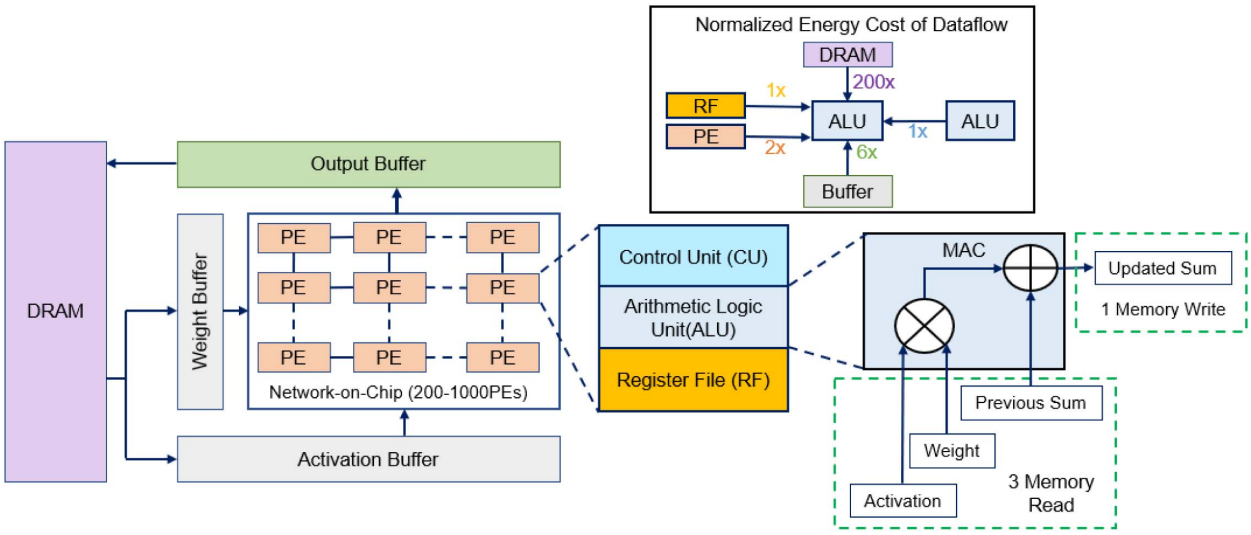


Fig. 16. *Block diagram of the generic DNN inference accelerator [2]. The PEs consist of CU, ALU, and RF. The main computation in the ALU unit is the MAC. Each MAC may require four memory read/write operations. The energy cost of each data flow in the memory hierarchy is also illustrated.*

and weights using RLC [250]. In addition, by zero skipping, the weights and MAC calculations can save 45% of energy.

E. Intel OpenVINO[1] Toolkit

Intel Open Visual Inference and NN Optimization (OpenVINO) optimizes the pretrained NN models for Intel hardware platforms to make the inference. The OpenVINO workflow consists of four main parts [258]: 1) model preparation and training; 2) model optimizer; 3) inference engine and tuning; and 4) deployment.

1) Model Preparation and Training: The OpenVINO toolkit supports most of the popular ML frameworks, such as Caffe, TensorFlow, ONNX (Pytorch and Apple ML), MXNet, and Kaldi. A DL model can be trained using these frameworks or downloaded from the Open Model Zoo. The Open Model Zoo is a pool of pretrained models for numerous applications, such as object detection, face recognition, pose estimation, text detection, and action recognition.

2) Model Optimizer: The model optimizer optimizes and converts the pretrained model to an intermediate representation (IR) suitable for inference. It is a cross-platform command-line tool to prepare the DL models for optimal execution in hardware. The model optimizer converts the trained model to an nGraph compatible IR (.xml + .bin files) suitable for inference. The optimizer removes the excess layers and operations, and implements pruning, freezing, quantization, fusion, and so on. An accuracy checker utility can be run to evaluate the accuracy of the inference. Further acceleration of inference is possible by quantizing the model to an INT8 precision using the posttraining optimization tool.

3) Inference Engine and Tuning: The IR is fed to the inference engine. The inference engine loads and compiles the optimized model to run the inference on inputs. Model compatibility is checked based on the training framework and inference hardware. It can execute the inference on multiple Intel[2] devices, such as CPUs, FPGAs, graphics processors, Neural Compute Stick (NCS) 2, and Movidius[2] vision processing units (VPUs). To trial and test the inference performance, the OpenVINO[2] Tuning Utilities and Inference Engine can collaborate. The benchmark utility can provide throughput and latency measures through iterative tests, and comparison among differently configured inferences can be obtained using the cross-check utility.

4) Deployment: The Deployment Manager is a command-line tool to assemble the tuned model, IR files, applications, and required dependencies into a runtime package for the target hardware devices.

F. MATLAB Deep Learning HDL Toolbox

The Deep Learning HDL Toolbox[2] in MATLAB R2021b (MathWorks Inc.) and the above versions support the inference on Xilinx and Intel FPGAs and SoCs [265]. This toolbox enables users to design customized DL models and evaluate the hardware realization performance. The DL model building and training are possible in MATLAB. The pretrained model from other frameworks can also be loaded into the MATLAB environment. The HDL toolbox supports 8-bit integer quantization of weights to improve throughput, memory, power, and computational requirements. Portable and synthesizable Verilog or VHDL code of the DL model for deployment on edge hardware

[1]Trademarked.

[2]Registered trademark.

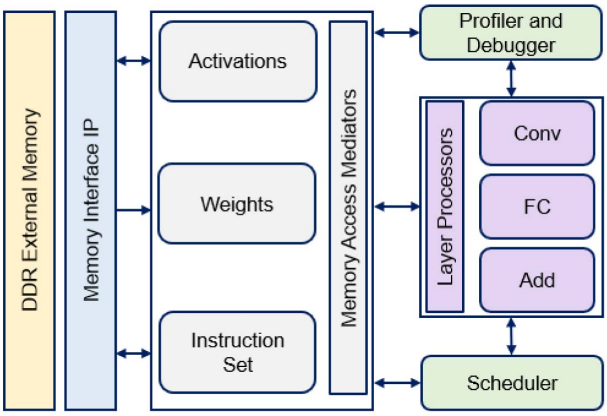


Fig. 17. *MATLAB HDL DL Processor IP [265]. The processor consists of four AXI4 master interfaces, DDR external memory, convolution, FC layer processor, activation normalization, and controller through scheduling logic.*

(FPGA and SoCs) can be generated using HDL Coder and Simulink.

One of the advantages of this toolbox is that the compile command automatically generates FPGA instructions without requiring manual reprogramming. The Ethernet or JTAG interface can be used to connect the FPGA and deploy instructions with the DL processor IP core. During predictions, the actual on-device performance metrics, such as layer-level latency, can be evaluated to identify performance bottlenecks. The synthesizable RTL from HDL coder can adapt to a variety of workflows and devices. An IP core with standard AXI interfaces can be created for the integration into SoCs.

The MATLAB HDL processor IP core is shown in Fig. 17. The toolbox includes a DL processor consisting of four AXI4 master interfaces, DDR external memory, convolution, FC layer processor, activation normalization, and controller through scheduling logic. Input data and parameters can be stored in the external DDR memory and transferred to block RAM using one of the AXI4 master interfaces. The block RAM provides activations to the generic convolution and FC processor. The convolution and FC processor perform the convolution operation and FC layers' computation. The activation normalization module serves the purpose of adding the ReLU nonlinearity, a max-pooling layer, or performing local response normalization (LRN). There are two controllers in the IP core: one for the convolution and the other for the FC layer. The generic convolution/FC processor and activation normalization can process one layer at a time. The next layer is processed using the controller (scheduling). The prediction results or scores can be transmitted back to MATLAB through the AXI4 Master interface and stored in the external DDR memory. This processor IP core is edge hardware platform-independent and, therefore, can be deployed to any custom hardware platform. The processor IP core is reusable and sharable to accommodate the DL model of different sizes.

The DL HDL toolbox provides hardware implementation of convolution (2-D convolution and depthwise convolution), FC layers, activation functions (ReLU, leakyReLU, and clipped ReLU), batch normalization, cross-channel normalization, dropout, pooling (max, average, and global average pooling), addition, and concatenation layers. It also supports a few layers from other frameworks, such as Keras and ONNX. The DL HDL toolbox supports Xilinx Zynq[1]-7000 ZC706 FPGA, Zynq UltraScale+[2] MPSoC ZCU102, and Intel Arria[1] 10 SoC devices. The MATLAB DL HDL provides inference capability on Intel or Xilinx FPGA for many popular pretrained CNNs (AlexNet, VGG, ResNet, DarkNet, YOLO V2, MobileNet-v2, GoogLeNet, and so on) for classification, object detection, and software-defined radio.

G. XCUBE-AI

X-CUBE-AI can automatically convert pretrained DL and classical ML algorithms to run inference on STM32 Arm[1] Cortex[1]-M-based microcontroller units (MCUs) [266]. It supports the validation and performance measurement of ML algorithms on STM32 devices without manual code generation. The workflow of X-CUBE-AI for DL inference on the STM32 MCUs is presented in Fig. 18. The pretrained models can be automatically converted to an optimized MCU-compatible C code.

X-CUBE-AI provides support for Keras, TF-Lite, and ONNX frameworks. Pretrained models from other frameworks need conversion to the standard format supported by X-CUBE-AI (.h5, .tflite, and .onnx). For DL model optimization, X-CUBE-AI supports 8-bit quantization. Weights can also be stored in external memories (flash and RAM) to make inferences for deeper networks. In addition, multiple models can be run on a single STM32 MCU.

H. XILINX Deep Neural Network Development Kit (DNNDK)

To provide efficient DL inference on Edge platforms, Xilinx developed the DNNDK. The DNNDK is a software development kit (SDK) that enables acceleration of the

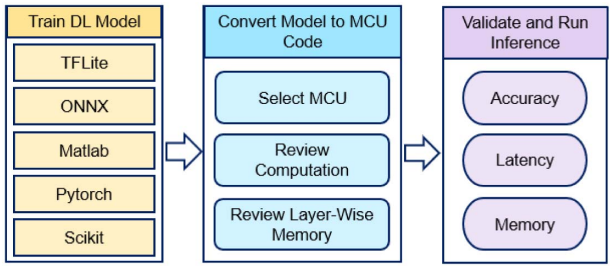


Fig. 18. *Integration of X-CUBE-AI to run DL inference on STM32 microcontrollers [266].*

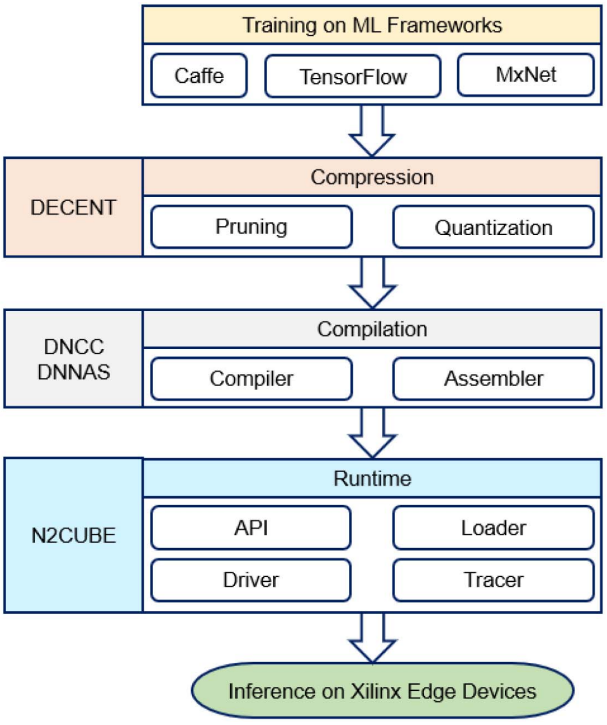


Fig. 19. *Xilinx DNNDK workflow and associated tools for DL inference on Xilinx edge devices [267].*

DL inference in programmable logic. The core component of the DNNDK is the DL processor unit (DPU). The DPU is scalaßith Xilinx[1] Zynq[1]-7000 series FPGA and Zynq UltraScale+[2] MPSoC [267]. Zynq and Zynq MPSoC are heterogeneous systems on chip with ARM processors with programmable logic. Using the lightweight C/C++ application programming interface (API) of DNNDK, inference application development is possible without prior knowledge of FPGA programming. It also includes an efficient tensor-level instruction set for CNN.

The DNNDK workflow and associated tools are shown in Fig. 19. To deploy the DL inference on the edge, the DNNDK follows five steps: 1) compression of the DL model; 2) compilation of DL models to generate ELF files for DPU; 3) programming DNNDK API to manage inputs and outputs, DPU kernel life cycle, and task management; 4) compilation of hybrid DPU application to make the CPU code and link to the ELFs for the DPUs; and 5) running the hybrid DPU application on the target edge device.

The constituent components of DNNDK include compression, compiler, assembler, runtime, simulator, and profiler. The Deep Compression Tool (DECENT) takes an FP32 pretrained model, weights, and a calibration dataset as input. A lightweight quantized model with an 8-bit fixed point representation is created. It supports coarse-grained pruning, posttraining quantization, and weight-sharing compression techniques. The DNN Compiler (DNNC[2]) is the compiler that converts the DL algorithms to IR and DPU instructions. It also maintains a balance between memory access and computational load to achieve maximum utilization of the DPU resources. The Cube of NNs (N2Cube) is the runtime engine for DPU that consists of a driver, DPU loader, tracer, and programming APIs. N2Cube provides loading of DNNDK applications, scheduling, and resource allocation. The DNN Assembler (DNNAS) compiles the DPU instructions into ELF binary code. The DPU profiler is composed of a DPU tracer and DSight. During DL inference runtime, the DPU tracer produces the raw profiling data. The DSight can generate the visualized charts for performance analysis using the profiling data. DNNDK v3.0 and above support both the Caffe and TensorFlow frameworks [267].

I. NVIDIA TensorRT

NVIDIA's TensorRT allows the optimization of DL models for faster inference in embedded and automotive devices. TensorRT takes a trained DL model as input and produces a highly optimized runtime engine to perform the inference. TensorRT is built on the parallel programming model CUDA and leverages sparse tensor cores [268]. It performs six optimizations for efficient inference: 1) reduce mixed precision; 2) layer and tensor fusion; 3) kernel autotuning; 4) dynamic tensor memory; 5) multistream execution; and 6) time fusion. In the training phase, the model parameters and activations use an FP32 representation. TensorRT calibrates the precision to convert FP32 into FP16 or INT8 precision. These quantization steps significantly reduce latency and model size. TensorRT uses KL-divergence to convert FP32 distribution into INT8. KL-divergence finds the difference between FP32 and INT8 representation to minimize the difference through an iterative search. The layer and tensor fusion merge the nodes in a kernel to optimize the memory. Such fusion reduces the cost of memory access for the tensor data for each layer. Kernel autotuning is the kernel-specific optimization that decides on the best layers, algorithms, and best possible batch size based on the target edge device. Memories are allocated to the tensor only for the duration of usage. It helps high-speed and efficient execution reducing the memory footprints and avoiding allocation overhead. TensorRT processes multiple input streams in parallel through Nvidia's CUDA stream. Time fusion is specifically for the RNNs that perform optimization over time steps with dynamically generated kernels. TensorRT-based inference can produce results up to 40× faster than the CPU-only platforms.

J. CEVA Deep Neural Network (CDNN)

CDNN is a graph compiler for pretrained DL model deployment on edge devices for efficient inference. CDNN incorporates network optimizations, quantization, data flow management, and libraries for both CNN and RNN [269]. CDNN consists of three main components: compiler, run-time software, and invite API. The compiler automatically optimized the pretrained models

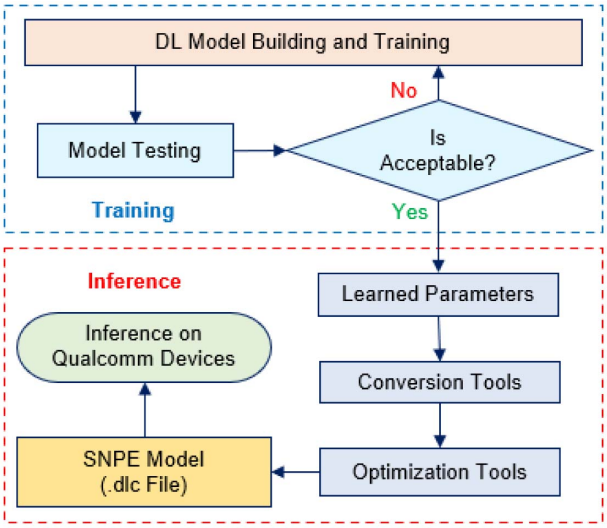


Fig. 20. *Workflow of Qualcomm[1] NPS [270] for DL inference acceleration in Snapdragon processors.*

through quantization. Run-time software package accelerates the inference in low-power embedded systems. The CDNN-Invite API provides network support and optimizations to ensure seamless incorporation and application of custom AI engines. The CDNN-optimized models can be embedded for SensPro, NeuPro, and CEVA-XM cores. The CDNN also incorporates layer fusion and various compression techniques to reduce memory bandwidth. It enables proper utilization of heterogeneous computing architectures and is flexible to split a network between multiple computing engines to ensure superior performance with minimal power consumption. For example, the AlexNet inference on CEVA-XM4 demonstrates 3× speedup, 1/30th power consumption of a GPU, and 1/15th memory requirement of a typical implementation. The CDNN fixed-point quantized AlexNet exhibited less than 1% accuracy degradation than the FP32 implementations on a CPU.

K. Qualcomm[1] Neural Processing SDK

The Qualcomm Neural Processing SDK (NPS) is designed to convert DL models trained on TensorFlow, Caffe, and ONNX to be compatible with Snapdragon[2] mobile platforms [270]. The NPS exploits the heterogeneous computing resources to run a pretrained NN on Kyro[2] CPU, Adreno[2] GPU, and Hexagon[2] DSP without connecting to the cloud. The NPS supports CNNs and custom layers. Fig. 20 shows the workflow of Qualcomm NPS. After building and training, the model files are transformed into a Deep Learning Container (.dlc) file to be processed by the Snapdragon neural processing engines (NPEs). After generating the model file, additional optimization is possible using quantization or compression techniques. An application can be designed to quickly execute the converted model to make the inference on mobile platforms. Qualcomm provides Qualcomm NPS C++/Java API or GStreamer plugins for ML application development and DL inference execution. After execution, the model performance and accuracy can be tuned using debugging tools.

L. Cadence Stratus HLS

The Stratus HLS developed by Cadence Design Systems Inc. can realize the DL inference in register transfer logic (RTL) [271]. It supports evaluating the performance, power consumption, and silicon area requirement for various DL designs. The working steps of Stratus HLS are illustrated in Fig. 21. The DL models are usually trained in other frameworks. The learned parameters (weights and biases) are extracted from TensorFlow or Caffe to calculate the inference in RTL. The same DL model architecture needs to be implemented in System C with parametrized datatypes and hardware constraints. Stratus HLS allows quantization of parameters ranging from 16-bit down to 12-bit fixed-point representation. From the System C architecture, the Stratus HLS can generate Verilog RTL. From the Verilog RTL architecture, the Joules RTL Power Solution is used to get an assumption on power consumption, Genus Synthesis Solution for area predictions, and Xcelium for throughput analysis.

M. Software Frameworks for Large-Scale Distributed Training

Distributed training facilitates large and complex DL models to train faster from the high volume of data by exploiting data and model parallelism. In data parallelism, data are split and distributed among available connected nodes, and the DL model is shared. On the other hand, DL models are split among nodes while using the same

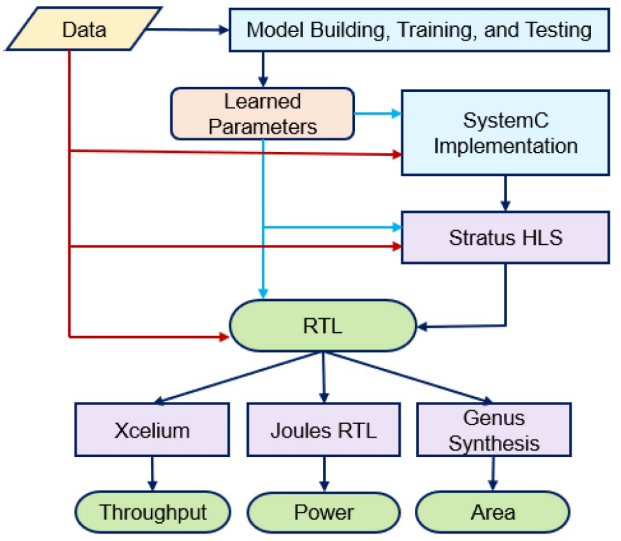


Fig. 21. *Cadence Stratus HLS for RTL synthesis and inference performance evaluation of DL models trained in TensorFlow [271].*

data in model parallelism. Data parallelism is more practical in most cases. Model parallelism is only feasible, while there are separately trainable model components that can be synced after training. There are software frameworks designed to achieve efficient distributed training. For instance, PyTorch and TensorFlow allow data parallelism by exploiting the distributed training modules.

DeepSpeed developed by Microsoft is built on PyTorch. It allows three-way parallelism (models, data, and pipeline) that facilitates memory and communication efficiency [290]. DeepSpeed enables high throughput and low latency for large DL models (having a trillion or more parameters) utilizing distributed computing resources. Other features include mixed precision training in single/multi-GPU/multinode, gradient accumulation, checkpointing, model parallelism, and compression. For instance, DeepSpeed can train up to 13 billion parameters on a single GPU without running out of memory, while PyTorch can handle only 1.4 billion [291]. In addition, the zero redundancy optimizer (ZeRO) facilitates memory efficiency by partitioning model states and gradients. In addition, 1-bit Adam [292], 0/1 Adam [293], and 1-bit LAMB [294] optimizers reduce the communication resource demand in DeepSpeed. Sparse attention kernels support long sequence input and sparse structures with faster execution and comparable performance. Inference-customized kernels' design and model quantization shrinks the DL model reducing the inference latency and cost. Transformer-based models, such as Megatron and HuggingFace, can make the inference efficiently using DeepSpeed model parallelism. Compression techniques, such as ZeroQuant [295], are designed, and many existing compression techniques are built-in in the DeepSpeed library for rapid compression, reduced model size, and faster inference.

Google introduced GPipe that combines data and model parallelism through pipelining [296]. GPipe library offers distributed training using synchronous SGD and pipeline parallelism. A DL model can be partitioned for parallel processing in different hardware accelerators and automatically split training minibatches into microbatches.

Horovod is another distributed DL training framework developed by Uber and hosted under Linux Foundation AI. It supports popular ML packages, such as TensorFlow, PyTorch, and Apache MXNet [297]. Horovod uses a message-passing interface (MPI) to run DL training in parallel, distributing the workload to computing facilities. A DL model computation can be distributed to hundreds of GPUs for parallel processing, scaling down the training time. For instance, a scaling efficiency of 90% is achieved for Inception-v3 and ResNet-101 in Horovod (128 servers, four pascal GPUs, and 25-Gb/s connectivity) execution [298]. In addition, Horovod can run on Apache Stark for data processing and model training in a single pipeline. It uses an optimized implementation of the ring-all-reduce algorithm called NCCL for efficient network bandwidth use while running a DL model over multiple GPUs.

Fairscale is an extension of PyTorch for efficient large-scale distributed DL training [299]. Fairscale allows DL model interlayer, intralayer, and tensor parallelism, memory and compute efficiency, and model splitting across multiple processing nodes. Mesh TensorFlow is a language that implements distributed computation graphs over TPUs using model parallelism [300]. TF replicator is another framework for leveraging distributed ML across TPUs [301]. High-level parallelism is obtained by first building subgraphs for all devices and connecting them by replacing the placeholders in cross-device computation. Therefore, the workload distribution can be scaled-up and switched between different accelerators. BigDL is a distributed DL framework for Apache Spark for data processing and DL in the end-to-end pipeline [302]. Ray is another high-performance distributed framework built on PyTorch for large-scale deep and RL [303]. In Ray, data are handled by sharing memory and hierarchical scheduling resulting in low-latency and high-throughput training. Ray is flexible in using different GPU clusters and compatible with most cloud providers.

Besides different software frameworks and libraries, cloud platforms support easy and efficient DL model training and performance evaluation. Google Colab, Amazon Web Services, and Microsoft Azure are a few prominent cloud computing infrastructures allowing quick DL model building, training, optimization, and inference at various capacities and scales [304].

XI. EDGE HARDWARE PLATFORMS FOR DEEP LEARNING INFERENCE

Several hardware devices are available to make the EI, such as FPGA, MCU, TPU, VPU, system-on-chip (SoC), and ASICs. With large amounts of data, available programming frameworks, and new algorithms, the DNN acceleration is contingent upon the computing capabilities of the hardware platform. For DNN acceleration, the graphics processing unit (GPU) or TPU is the popular choice in the cloud. Although GPU and TPU can provide high throughput, the associated energy cost is unacceptable for resource-constrained edge devices. Thus, customized neural accelerators are necessary for inference on edge devices. Such accelerators can achieve satisfactory results through architectural, memory, and dataflow optimization. A comparison of different choices of edge hardware platforms is presented in Table 6. The right choice of hardware is contingent upon the design specification in terms of latency, area, energy, cost, and flexibility.

A. General Purpose CPU and GPU

The general-purpose CPU contains billions of transistors and can effectively run DL training and inference. They contain powerful cores and high memory capacity that can handle vast amounts of operations. The

Table 6 Edge Hardware Comparison in Terms of Quality Metric

Quality Metric	Lowest → Highest			
Reconfigurability	ASIC	FPGA	GPU	CPU
Speed	FPGA	ASIC	CPU	GPU
Cost	CPU	GPU	FPGA	ASIC
Performance	CPU	FPGA	GPU	ASIC
Time to Market	CPU	GPU	FPGA	ASIC
Power	ASIC	FPGA	CPU	GPU
Area	ASIC	FPGA	CPU	GPU

CPU-based inference is not feasible on-edge due to high power consumption, low throughput performance, costly design, complicated manufacturing process, and large silicon area due to many redundant components. Moreover, the CPU does not fully exploit the parallelism possible for DL simulations. The GPU offers HPC using multicore architecture and leveraging parallel processing. CPU and GPU are software programmable allowing straightforward adjustment to run new or customized models. In contrast to the CPU, GPU comprises thousands of ALUs enabling parallel execution of a massive number of simple operations. GPUs are performance efficient but infeasible for EI due to energy constraints. The GPU power consumption is not tolerable for most smartphones, drones, and many other edge devices. Although there is high throughput performance, the limitation of GPU is the IO latency for transferring data and memory. Therefore, the usage of GPU for EI depends on the power and energy handling capacity of the target. The Jetson TX2 includes embedded GPUs that consume around 15-W power to run the DL inference [253]. The NVIDIA Jetson Nano development board consists of a 128-core GPU and quad-core ARM CPU that can smoothly run most NN backends and frameworks using NVIDIA Jetpack SDK.

B. System-on-Chip and Application-Specific Integrated Circuit

Researchers around the globe are focusing on ASICs and SoCs for DL applications due to the highest performance and energy efficiency. ShiDianNao [286] is a custom ASIC with improved memory accesses for low latency and power consumption. It is one of the accelerators from the DianNao [287] family targeting the embedded devices. UNPU [124] is an ASIC designed in a 65-nm CMOS process capable of running inference for convolutional, recurrent, and dense layers. It supports variable bit-width precision from 1 to 16 bit for weights and demonstrates that LUT-based serial processing improves the energy efficiency of the MAC operation. For different bit weight precisions, UNPU can achieve the peak performance from 345.6 to 7372 GOPS. Envision [218] targets the always-on-vision application for wearable devices. It reuses the inactive arithmetic cell for computation at a lower precision. Designed in a 28-nm CMOS process, this chip can achieve an energy efficiency of 10 TOPS for CNN processing for visual recognition. An ASIC for object tracking using pipelined direct feedback alignment (PDFA) is presented in [308]. It is capable of processing 34.4 frames/s with an energy efficiency of 1.32 TOPS/W. SNAP [309] is another ASIC designed in a 16-nm process that adopted unstructured sparsity for optimization. Within a 2.4-mm^2 chip area, it can achieve an energy efficiency of 21.55 TOPS/W for convolutional layers, and it can run pruned ResNet-50 inference within a power consumption of 348 mW. STICKER [310] is an ASIC for efficient inference of DL models containing convolution and dense layers. The highest energy efficiency of 62.1 TOPS/W within the 7.8-mm^2 chip size can be reached. It uses an automatic circuit to switch between nine sparsity modes and multimemory access modes to ensure better energy efficiency. FlexFlow [311] is a scalable accelerator that exploits the parallelism of computing platforms and CNN architectures to support a flexible data flow and data reuse.

Cadence Design Systems, Inc., developed the deep neural-network accelerator (DNA 100), an IP SoC for on-device DL inference [305]. DNA 100 supports ML frameworks, including Caffe, TensorFlow, TF-Lite, and so on. It can integrate end devices, such as wearables, AR/VR headsets, drones, always-on-vision cameras, smartphones, drones, robots, and IoT devices for numerous applications in computer vision, speech processing, and communications. It also supports the Android NN (ANN) API for on-device inference in Android devices. Current DNA 100 processors can run NN layers, such as convolution, FC, LSTM, and pooling. A single accelerator can deliver 0.5–12 TMACs, and multiple DNA can be stacked to achieve 100 s of TMAC/s.

Synopsis developed an intellectual property (IP) processor core for computer vision applications on embedded devices named DesignWare EV6x [306]. The accelerator allows 8- and 12-bit quantization. DesignWare supports OpenCV , OpenCL^2 C, and OpenVX^2 toolkit and consists of a 32-bit processor, an optimized CNN accelerator, and a 512-bit vector digital signal processor (DSP). When implemented in 16-nm FinFET processes, it can offer up to 2000-GMAC/s/W power efficiency.

Ceva Inc. has developed IP cores for NN processing named NeuPro-STM [269]. It consists of a CEVA-XM digital vision signal processor and a NeuPro-S engine for layer computations, such as convolution, activation, and pooling. This IP core also supports 8- and 16-bit quantization, sparsification, weight decompression, and multicore scalability. The processing capability of this accelerator ranges from 2 to 12.5 TOPS per core. It is scalable to use multicore processing for a performance boost of up to 100 TOPS.

Table 7 Summary of ASICs and SoCs for Edge Inference Targeting Various Use Cases

ASIC	Energy efficiency ($TOPS/W$)	Area (mm^2)	Process (nm)	Power Consumption (mW)	Supported DL Models	Target Applications
DNA 100 [305]	3.4 $TMACs/W$		16	850	All neural network layers	IoT, AR/VR, autonomous driving, surveillance, mobile devices
Stripes [123]	-	122.1	65	-	CNN (LeNet, AlexNet, GoogleNet, VGG, etc.)	Image classification
Eyeriss [250]	0.1265	12.25	65	235-332	CNN, AlexNet	Image analysis, computer vision
Bit Fusion [126]	-	5.93	16	895	CNN, RNN	Image classification, object detection, language modeling, optical character recognition
DesignWare [306]	2,000 $GMACs/W$		16		CNN, R-CNN, AlexNet, VGG16, GoogLeNet, ResNet, SqueezeNet, and YOLO	Object detection, classification, segmentation
UNPU [124]	3.08	16	65	3.2-297	CNN, RNN, FNN, VGG 16	Image classification
Envision [218]	0.26-10	1.87	28	7.5-300	CNN, VGG 16, AlexNet	Always-on-vision, face recognition
Lightspeeur 2803 [307]	24	81		700 (for 16.8 TOPS)	VGG, ResNet, MobileNet	Image classification, Object Detection and Tracking NLP, Visual Analysis
Lightspeeur 5801 [307]	12.6	36		224 (for 2.8 TOPS)	VGG, ResNet, MobileNet-v1, custom networks	Image classification, Object detection and tracking, NLP, Visual Analysis
PDFA [308]	0.7708-1.321	5.76	65	168	CNN, FCN	Object tracking
SNAP [309]	21.55	2.4	16	16.3-364	CNN, FCN, ResNet-50, AlexNet, VGG-16	Image classification
Cambricon-X [233]	-	6.38	65	954	LeNet-5, AlexNet, VGG, etc.	Classification
Scalable processor [119]	0.3-2.6	2.4	40	25-288	CNN, AlexNet, LeNet-5	Image classification
Sticker [310]	0.411-62.1	7.8	65	20.5-248.4	CNN, FCN, AlexNet	Object detection
EIE [243]	138,927 $Frames/J$	40.8	45	600	CNN (AlexNet, VGG-16), RNN, LSTM	Image classification, object detection, automatic image captioning
FlexFlow [311]	0.38-0.5	3.89	65	1000	CNN	Image analysis
SqueezeFlow [249]	-	4.80	65	536.09	Sparse CNN (VGG 16, AlexNet, GoogleNet)	Image analysis

NeuPro-S targets applications for advanced autonomous driving, ADAS, AR, smartphones, UAVs, and cameras.

Gyrfalcon Technology Inc. (GTI) has developed the LIGHTSPEEUR[1] [307] neural accelerators targeting the edge devices for various applications, such as IoT, mobile edge computing, portable devices, cameras, AR/VR products, NLP, and autonomous driving. The intelligent matrix processor Lightspeeur 2801S uses memory for neural processing based on APiM architecture. Eliminating the data movement enables a superior energy efficiency at 9.3 TOPS/W. The architecture features on-chip parallelism, on-chip computing, and reduced memory bottlenecks for AI inference. Lightspeeur 2802M is designed on a matrix processing design that uses magnetoresistive random access memory (MRAM) technology. It enables multitask performance with multimodel storage on a single chip delivering 9.9 TOPS/W. Lightspeeur 2803 is the dedicated CNN inference engine for audio and video processing. It can process 24 TOPs/W, 16.8 TOPS at 300-MHz speed, and 700-mW power within a small size of 9 mm $\times$ 9 mm. Models can be cascaded to allow multiple chips to support larger models. The Lightspeeur 5801 delivers the neural processing of 12.6 TOPS/W and 2.8 TOPS at 200-MHz speed and 224-mW power consumption.

The Coral edge TPU [312] developed by Google is an ASIC for EI. The Edge TPU is primarily capable of inference acceleration and certain lightweight transfer learning. The advantages of edge TPU are low power consumption, cost-efficiency, NN processing at high speed, and off-line capabilities. Coral TPU supports ML frameworks, such as TF-Lite, and DL models, such as YOLO and R-CNN.

The ASICs are the most efficient hardware to realize the EI. Table 7 provides an overview of several successful SoCs and ASICs for a variety of applications. However, most of the computations realized in ASIC are hardwired in silicon. DL networks and optimization algorithms are continually being changed and enhanced. Hence, deploying an ASIC solution for inference is expensive due to reduced flexibility and limited programmability. Introducing flexibility in the ASIC architecture increases its silicon area resulting in

increased cost and power consumption. Thus, coarse and fine-grained reconfigurable architectures are other choices for the rapid prototyping of EI.

C. Field-Programmable Gate Array and Reconfigurable Architectures

The spatial architecture of the hardware accelerator for DL models can be of two types: coarse-grained and fine-grained. FPGA is a fine-grained reconfigurable architecture that is programmable through hardware description languages (HDLs). The coarse-grained reconfigurable accelerators (CGRAs) have evolved as an architecture for EI due to their capability to operate at higher clock frequencies, faster compute performance, reduced compilation, and reconfiguration time. For efficient tensor processing of ML models, coarse-grained spatial architectures are a popular selection.

1) FPGA: A reconfigurable architecture can be designed faster than ASICs to easily adapt to model changes. FPGA contains programmable logic blocks and configurable interconnections. FPGAs are more flexible in architecture than ASICs due to fast computation and reconfigurability. HDLs (e.g., Verilog or VHDL) are used to define the connection between digital hardware components. FPGAs are easy to use for fast customization and performance evaluation for DL inference on hardware. Due to the increased demand for EI, small and high-density FPGAs are now being considered [11], [315], [316], [317]. The ZYNQ7020 FPGA is one of the popular reconfigurable architectures exploited to implement and prototype CNN inference. For example, VGG-16 CNN implementation with 8-bit fixed point quantization on ZYNQ7020 FPGA achieved a performance of 84 GOPS [315]. Similarly, a 16-bit fixed-point quantized CNN implementation with 13 GOPS has been demonstrated in [316]. Lite-CNN is a configurable implementation of the INT8 quantized CNN in ZYNQ XC7Z020 FPGA. It can accomplish 408 GOPS peak performance and 33 GOPS/W energy efficiency. One of the challenges of FPGA-based inference is the conversion of trained models to FPGA interconnections. Other challenges include slow configuration, long compilation time, and low clock frequency. Notable works on DL inference on FPGA are presented in Table 8 with the implemented model, peak performance, and energy efficiency.

2) Reconfigurable Architecture: The general architecture of the DL inference accelerator consists of an array of interconnected PEs, RFs, and shared/scratchpad memory. Low latency and high energy efficiency can be achieved with optimized PE design, effective memory access control, spatial data reuse, and optimized computations. Such coarse-grained architectures are more power-efficient for DL inference execution compared to the CPU and GPU cores [250], [318], [319]. Tensor processing of DL models is efficiently processed using the PEs. With local RFs and memory, high spatial and temporal data reuse can be achieved by minimizing off-chip memory access. Therefore, using coarse-grained spatial architecture can achieve low latency, high throughput, and energy efficiency for DL inference acceleration. A summary of representative research on reconfigurable architectures for DL EI is presented in Table 9 consisting of supported models/layers, peak performance, and energy efficiency.

Table 8 State-of-the-Art DL Inference on FPGA

FPGA Platform	DL Model	Peak Performance ($GOPS$)	Energy Efficiency ($GOPS/W$)
Xilinx ZC702	XNOR-Net [140]	207.8	44.2
Xilinx ZC702	DoReFa-Net [143]	410.2	181.5
Zynq ZU3EG	Synetgy [164]	47.09	8.56
Arria 10 GX1150	BBS-LSTM [96]	304.1	15.9
Xilinx XC7Z100	DeltaRNN [313]	192.0	26.3
Xilinx Virtex-7	C-LSTM [314]	131.1	6.0

Eyeriss [250] is one prominent CGRA for convolution implementations. It contains 168 PEs connected to the network-on-chip (NOC). Eyeriss uses data compression for the chip, external memory, and sparsification in intermediate layers. With a 16-bit fixed-point quantization, the architecture achieved 166 GOPS/W with an average power consumption of 278 mW to get inference results for AlexNet. Eyeriss [250] adopted two major optimizations: 1) data reuse to limit memory access and 2) exploiting data statics to improve computations and prevent unnecessary memory reading. Thus, with a minimum data movement to external memory, improved energy efficiency is achieved.

An improved version of Eyeriss architecture is the Eyeriss v2 that generalizes the Eyeriss for compact and sparse DL architectures [237]. The Eyeriss v2 architecture is presented in Fig. 22. Each architecture consists of an array of PEs and GLBs. The constituent PEs and GLBs are grouped in Eyeriss v2 to adopt a hierarchical architecture minimizing the communication overhead.

The Thinker chip [320] supports three levels of reconfigurability: 1) the PE can be constructed with 8- or 16-bit multipliers; 2) support for zero skipping; and 3) improved bandwidth distribution configuring on-chip memory. In addition, parallelism is fully exploited to prevent idle PEs. The Thinker chip can achieve 590-GOPS/W energy efficiency within 335-mW power consumption in a 16-bit configuration.

The Sparse CNN (SCNN) [244] performs sparsification in both activations and weights. The SCNN architecture is reconfigurable consisting of a multiplier array and a set of accumulation registers. In [321], a reconfigurable ASIC implemented in the 65-nm process is presented targeting hybrid NNs. The PEs consist of two 8-bit multipliers, which can be used independently or jointly for 16-bit operations. It supports variable bit-width quantization. This

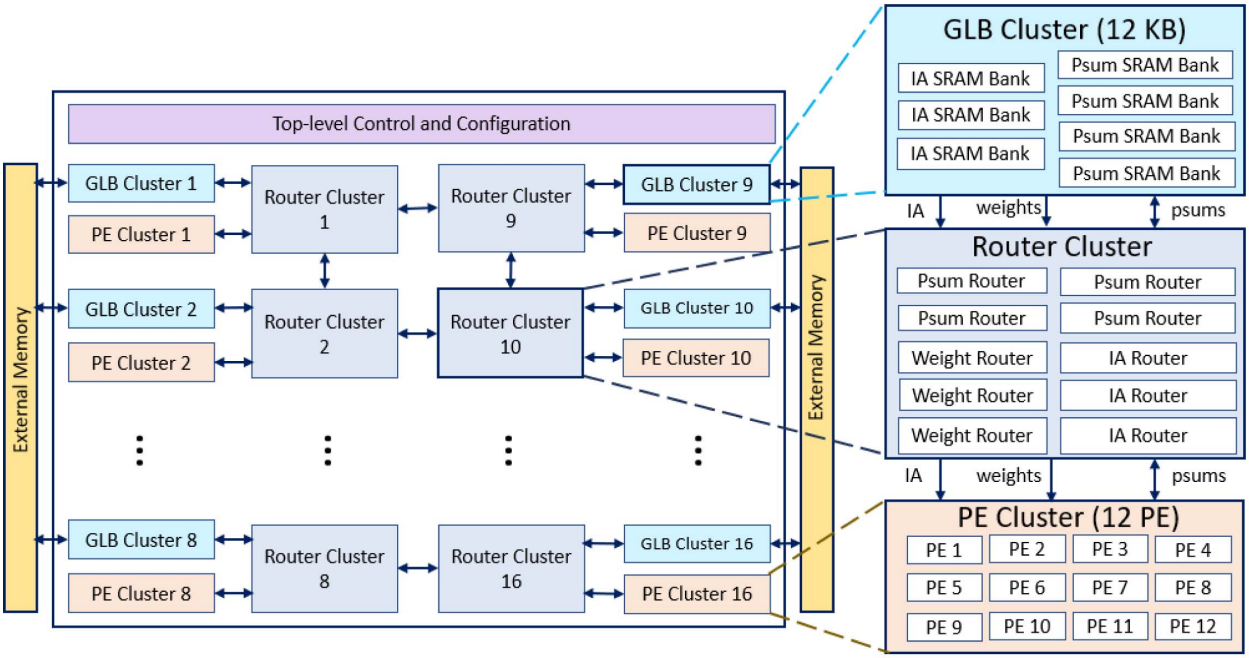


Fig. 22. *Eyeriss v2 high-level hierarchical architecture [237]. The PEs and GLBs are arranged into clusters supporting NOC flexibility. The architecture consists of 16 PE clusters and 16 GLB clusters. Each PE cluster consists of 12 PE, and each GLB cluster contains 12 kB of memory.*

architecture is capable of handling convolution, recurrent, and dense layers with supports for pooling and nonlinear activations with a peak performance of 409.6 GOPS and 5.09 TOPS/W energy efficiency.

The DNPU is another CGRA processor [322] that supports CNN and RNN processing. For dense and convolution layers, the architecture has dedicated units. The pooling and activation functions are executed by a central module. Matrix multiplications for FC layers can be configured as 4-, 8-, or 16-bit integer arithmetic. With 4-bit multipliers, the DNPU architecture can achieve a peak performance of 1.2 TOPS and an energy efficiency of 3.9 TOPS/W.

The SDT-CGRA [323] architecture includes up to three MAC instructions. A programmable delay is included in the PEs to map temporally closer data. The custom stream buffers are programmable in a very long instruction word (VLIW) approach that accelerates the DL access patterns.

Tianjic [324] and Simba [325] are two spatial architectures that utilized the concept of distributed weights [326] and weight exchange among PEs [319]. The PEs are decentralized with preloaded weights that remain stationary throughout the inference execution. The activations can propagate inter- and intra-PEs for computation.

A dynamically reconfigurable processor (DRP) is a CGRA core to accelerate inference on embedded devices [327]. The core consists of an array of dynamically reconfigurable PEs. It supports large networks by adjusting the architecture for different layers at runtime. DRP supports 16-bit integer or floating-point quantization and binarizations.

For RL, a CGRA architecture is introduced in [328]. The PEs support addition, multiplication, or a combination of both in a stationary fashion. Different activations can be chosen using the configuration register. The reconfigurable cells include address generations inside the architecture, and data can be locally stored. Reinforcement training is controlled using global communication lines.

The neural processing unit (NPU) is presented in [329] for the project Brainwave to provide real-time AI. The core NPU is a microarchitecture that is spatially distributed and capable of performing 96k MACSs. The architecture provides a custom SIMD instruction set for programming and can use vectors and matrices as datatypes. An RNN execution on Intel Stratix10280 FPGA using the Brainwave NPU has demonstrated a peak performance of 10–35 TOPS.

For efficient processing of DL inference, the neural accelerators exploit parallel processing and memory access. There are three main sources of data to be processed in

Table 9 Summary of Reconfigurable Architectures for DL Inference

Reconfigurable Architecture	Peak performance ($GOPS$)	Energy Efficiency ($GOPS/W$)	Supported DL Models
Eyeriss [250]	46.2	166	CNN
Eyeriss v2 [237]	153.6	962.9	CNN
Thinker chip [320]	409.6	1060	CNN, RNN
DRP [327]	960	-	CNN, DNN
Lite-CNN [317]	363	33	CNN
DNPU [322]	1214	8100	CNN, RNN
NullHop [248]	450	3000	CNN
SDT-CGRA [323]	92.3	50.78	CNN, k-means, PCA, SVM, etc.

the neural accelerators: weights, IAs, and the psum at the output of the activation functions. The NOC can increase data reuse, thus reducing memory footprint through direct information exchange between PEs. The usage of the GLBs can reduce the off-chip DRAM access time, thus improving the latency. Data prefetch using the double buffer technique in the GLB can fasten the computation even more [320].

D. Vision Processing Unit

The VPU is a customized microprocessor chip for EI for numerous machine vision uses. The VPUs are optimized to provide inference results for pretrained CNNs and other computer vision algorithms at low power with minimal performance degradation. Successfully deploying the vision processor can be used for numerous use cases in security, robotics, smart homes, drones, IoT, always-on-vision cameras, VR, AR, smartphones, and wearables.

Movidius Myriad X is a vision processing SoC that supports 8- and 16-bit integers and 16-bit floating-point quantization. It consists of a 4k image processing pipeline that can take input from up to eight HD sensors connected to the processor. With parallel processing and minimized data movement, it can achieve a peak performance of up to 4 TOPS for image, vision, and DL tasks. It is programmable with the Myriad Development Kit (MDK) and supports the Intel OpenVINO toolkit.

Intel NCS 2 is a small fanless USB vision processor developed on the Intel Movidius Myriad X processor. NCS 2 supports most of the popular ML frameworks, such as TensorFlow, Caffe, Apache MXNet, PyTorch, and ONNX. The USB containment of the chip makes it compatible with Raspberry Pi and Intel Next Unit of Computing (NUC) devices. NCS 2 can make EI for image processing and computer vision applications, NLP, machine translation, and so on.

The neuFlow [288] is a general-purpose vision processor for real-time object detection, recognition, and localization. This vision processor implemented on Xilinx Virtex 6 FPGA demonstrates 10 W of power consumption and is capable of processing 12 frames/s. The performance analysis has shown a 100× speedup compared to a laptop for segmenting 20 categories of the object from street scenes.

The pixel visual core (PVC) [330] from Google is a coprocessor for image analysis incorporating an ARM Cortex-A53. The PVC is an optimized architecture to minimize the power consumption while idle but requires a power budget of 6–8 W for a short burst of around a few seconds while operational. It supports the TensorFlow framework and can perform 3.28 TFLOPS at 800 MHz. Each arithmetic logic unit (ALU) consists of 256 PEs that are arranged as 16 × 16 2-D arrays. The PVC architecture consists of eight image-processing custom cores each having 512 ALU. PVC supports 8-, 16-, and 32-bit integer arithmetic operations.

Mobile EyeQ[1] [331] is another SoC for vision processing at low power targeting the ADAS and autonomous driving applications. The fifth-generation processor EyeQ5 is implemented on the FinFET 7-nm process. It is a VLIW SIMD processor having flexible memory access and multicore capability, and can perform 24 TOPS within 10-W power consumption.

E. Microcontroller Unit

MCUs are devices that work as an essential part of embedded systems. Therefore, DL capabilities in MCU can bring intelligence to billions of products from industry, households, automobiles, medical equipment, vision, and so on. Some of the advantages of EI on MCU include low power consumption, preserving privacy, easy installation, and low cost. Often MCU is a good choice for a controller rather than a computational job. However, advanced MCUs with complicated logic control, high operating speed, and large cache can make efficient DL inference in low latency. Lightweight DL models without FLOPs are suitable for MCUs. Therefore, fixed-point quantization and structured pruning are the prominent compression techniques used to deploy the DNN models on MCU. Modern MCUs support 8- and 16-bit fixed-point operations with some parallel computing capability. The limited memory capacity is one of the major limitations of MCUs.

CMSIS-NN [159] is a DL library targeting the ARM Cortex-M microcontrollers. CMSIS-NN consists of important DL kernels and supports converting pretrained models and posttraining quantization. The parallel ultralow-power (PULP-NN) library [332] can help to deploy DL on GAP8 processors. The GAP8 processor includes software-controlled scratchpad memory improving efficiency for memory-intensive DL inference. As most of the existing MCUs do not support SIMD instructions, the available instruction set architecture (ISA) poses limitations to running quantized DL models on MCU. Even if the parameters and data are quantized, the computation must be performed one by one using the 32-bit registers of MCU. In [333], an extension of the RISC-V ISA has been proposed that supports subbyte quantization. One limitation is that RISC-V microcontrollers are not widely available in the market, hence requiring expensive custom manufacturing. Hello Edge [334] implements various DL models on MCU for keyword spotting for speech-based smart devices. The KWS system is required to be always on with minimal power consumption to confirm the longevity of the battery. Besides, high accuracy and low latency are two critical requirements to ensure a satisfactory user experience. Hello Edge presented the implementations of CNN, RNN, convolutional RNN, depthwise separable CNN, and others on ARM Cortex-M microcontrollers with performance analysis.

TF-Lite has extensions for running DL inference on memory constraint MCUs [258]. For example, the ARM Cortex M3 having only 16 kB of memory can run many

basic ML models. TF-Lite supports 32-bit MCU architecture, such as the Arm Cortex-M series, ESP32 architectures, and development boards.

MicroAI [335] is another DL framework targeting end-to-end training, quantization, and deployment on 32-bit MCUs. This framework has been evaluated on speech recognition, activity recognition, and image classification on Ambiq Apollo3 and STM32L452RE microcontrollers showing competitive memory and energy efficiency.

Limitations of microcontrollers include less memory, slow processing compared to CPU or GPU, lack of parallelization, low clock frequency, and so on. Besides, MCU relies on general-purpose processors that cannot support vectorization or thread-level parallelism.

F. Emerging Memory-Based Neural Accelerator

The digital neural accelerators split the matrix-vector multiplication (MVM) into numerous MAC operations to perform the operations cyclewise. In the processing-in-memory (PIM) architecture, the emerging nonvolatile memory (eNVM) is exploited to realize the MVM operations of DNN efficiently. For instance, using RRAM [336], PCRAM [337], and memristors, the MVM operation can be performed fast in the analog domain within one clock cycle. However, the limitations are the requirements of analog-to-digital converter (ADC) and digital-to-analog converter (DAC) that constitute around 50% of the power consumption of the overall system. The traditional SRAM [338], DRAM [339], and Flash [340] memories also can be modified to realize the PIM implementation of DNN algorithms. The PIM technologies are still in their infancy and are being extensively studied in academia rather than industry adoption.

A BNN requires only two stable states. Therefore, BNN can be realized easily by utilizing the memory architectures and logic operations [341]. Irregular zeros in the weight matrix generated from quantization impose a challenge on execution speedup and energy saving due to crossbar processing. The weight matrix can be decomposed into smaller submatrices to be mapped onto a crossbar as done in SNrram [342]. The weight matrix can also be decomposed into blocks to cluster the zero columns for coarse-grained pruning [343], [344]. Fine-grained crossbar processing is demonstrated in [345]. Instead of activating the entire crossbar at once, a matrix block named operation unit (OU) is activated every time [345]. The zero inputs are skipped, and nonzero elements are grouped to activate the corresponding crossbar. As a result, more operations can be realized by the same crossbar within fewer clock cycles, thus improving the inference performance.

XII. EDGE TRAINING AND INFERENCE IN DISTRIBUTED ENVIRONMENT

DL requires substantial memory, computations, and communication resources to execute the training and inference, which is often impractical for a single-edge device. Therefore, distributing the training and inference task to different edge devices can reduce the burden on individual devices. Through collaboration and shared models, such distributed schemes collectively can achieve better performance. For distributed training and inference, novel algorithms are being emerged, which are summarized in this section.

A. Distributed Edge Training

Edge training is performed in proximity to the data collection for cost-effective communication in data transfer and ensuring user privacy by processing on-edge. The edge training can be performed in a single edge device or server if the computational capability and need match. For instance, DL models are trained in a smartphone for activity recognition and audio signal sensing in [346]. Similarly, researchers also investigated DL training in wearable [347] and embedded [348] devices. However, two major constraints in running training on single-edge devices are the memory constraints and the computational need for larger DL models [349]. Therefore, edge training is mainly performed in distributed manner learning from data in different edge devices and servers.

DeepCham [350] consists of one edge server connected to multiple mobile devices. The large model is updated by aggregating the parameters from edge nodes, and the mobile devices are trained for object recognition in local visual domains. Scheduling schemes for edge training are presented in [352], where the edge devices upload a training profile requesting cloud access. Depending on availability, the cloud server assigns edge servers to perform the training. Peer-to-peer distributed training is another infrastructure in which edge devices are given equal priority [353]. Each edge performs partial training on the local data and exchanges the parameters with connected nodes. This approach has demonstrated comparable performance to centralized cloud training for activity and pattern recognition tasks [353]. The concept of transfer learning has also been exploited in edge training [354], [355]. For example, features learned in a trained model are transferred to edge nodes that are updated by retraining with local data [354]. In addition, memories shared among edge devices can reduce the storage burden of any single-edge device. Besides, model compression to 8-bit fixed precision quantization [355] results in faster computation.

Interactive ML (iML) involves users, providing information to the learning process and monitoring the output in subsequent iterations [356], [357]. In [358], iML is exploited in edge training for human activity recognition (HAR) and demonstrated that a few training iterations achieve acceptable performance.

Federated learning is an emerging technique of edge training using data from multiple nodes [359]. A shared model can learn collaboratively from local training data instead of requiring to accumulate the data to a central cloud server [360]. There are two architectures of

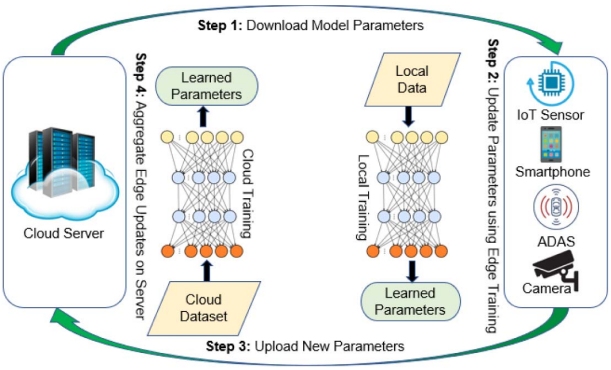


Fig. 23. *Federated learning scheme of edge training. Learned parameters are downloaded into the edge that is updated using local data collected from sensors. New parameters are aggregated into the cloud, and refined models are propagated to edge devices [351].*

federated learning, such as cloud-based and edge-based. An overview of cloud-based federated learning is illustrated in Fig. 23. The untrained DL model from the central cloud server is assigned to different edge devices to accomplish the training using the local data. After local training, the model updates are aggregated in a central cloud server and usually averaged [361]. The changes in the shared model are downloaded to edge devices. In this process, the training data remain in the edge devices. Only changes in learned models are uploaded to and downloaded from the central cloud server. Therefore, it maintains data and user privacy, and saves communication resources. In edge-based federated learning, an edge server serves the functionality of a central cloud server. Besides, a hierarchical approach can include edge and cloud servers where edge servers assign the training to edge devices.

Gradient optimization of the shared model considering local updates and maintaining connectivity remains challenging in federated learning. To mitigate the gradient optimization problem, selective SGD [362] has been proposed, which allows independent training on edge on local data and selective sharing of parameter updates to the central server. An extension of this approach is FedAvg [359], which considers the unbalanced and non-independent identically distributed (non-i.i.d.) data through iterative model averaging. In this method, the edge devices update the central aggregator with one-step SGD, which is averaged in the central server. In federated learning, each connected edge device transfers the model updates to the central server. If the model is larger, it poses a serious bottleneck to the communication resources. There are research efforts to minimize the number of update rounds in federated learning to ensure communication efficiency [359], [363], [364].

The DL model splitting is another edge training approach that transfers the processed data instead of raw data to the cloud server providing enhanced user privacy. The DL models are trained at both the edge device and server by partitioning the model into parts. However, determining the partition point is challenging. Researchers proposed different techniques for appropriately performing the DL model splitting, such as differential private mechanism [365], Arden (privAte infeRence framework based on DNNs) [366], and hybrid edge-cloud framework [367]. As distributed edge devices are exploited in edge training, parallelism in data and model can facilitate computation load balancing. For instance, PipeDream is model parallelism through pipelined training presented in [368]. The minibatches are simultaneously injected into the connected systems to ensure parallel utilization of computing resources and automatically determine the DL model splitting that speeds up the training.

A factorization-aware training is presented in [369] for natural language understanding (NLU) in resource-constraint devices, such as voice assistants. Joint training of compressed transformer model, DistilBERT, exploiting factorization of linear mapping has demonstrated around 84% model size reduction with approximate 10% accuracy degradation.

B. Distributed Edge Inference

Edge devices have limited resources which can cause slow DL inference. Therefore, splitting the inference workload and distributing it among connected edge devices has shown promise [19]. Distributed EI can be of two types: vertical in different levels of the edge device, edge server, and cloud server; horizontal among multiple devices on the same level. For instance, mobile deep inference (MODI) is a vertically distributed inference framework [370]. Multiple DL models in both compressed and uncompressed formats are stored in a central server that dynamically decides the deployment platform. For instance, if resources are limited, a compressed model is employed, and if sufficient resources are available, uncompressed models are deployed. In addition, MODI facilitates joint inference in edge devices and edge servers. DL models are mapped across different computing hierarchies: Distributed DNNs (DDNNs) [214]. Part of data analytics is performed in edge devices reducing the communication overhead. In addition, early exiting at vertical exit points results in faster inference.

MoDNN [371] is a horizontally distributed platform for DL inference in resource-constrained edge devices. In MoDNN, the pretrained DL models are used and layer scanning identifies layer types. The model partitioning and edge devices are determined based on layer types. Empirically using the VGG-16 model on the ImageNet dataset employing two to four mobile devices resulted in a 2.17—4.28× inference speedup. A communication-aware distributed inference across six edge devices is presented in [372] for YOLOv2 inference. DL inference in mobile robots utilizing the optimal model partitioning to incorporate idle robots in proximity shows a 6× speedup than remote execution in the cloud [373].

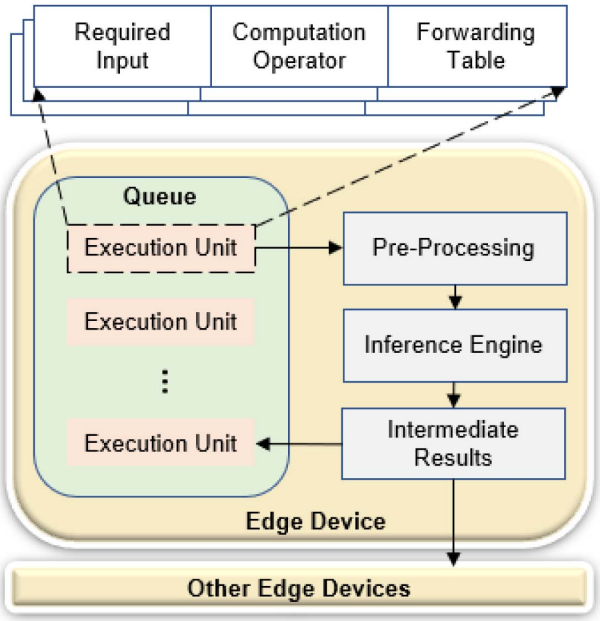


Fig. 24. *Overview of the distributed inference system in EdgeFlow [380].*

Model parallelism can distribute the inference workload in a sequential or parallel manner. In sequential distribution, DL layers are assigned to different edge devices [25], [374]. For instance, input is processed by one device and passed results to the next for further computation. However, sequential approaches often keep the devices idle if enough inference computation is not in the queue. In parallel distribution, feature dependence of different layers is exploited. For example, in DeepThings [375], feature maps are partitioned into nonoverlapping tiles using fused tile partitioning (FTP) to assign them to different edge devices. A scheduling process is employed to ensure data reuse and reduce inference latency. An inference speedup of 1.7—3.5× using two to six edge devices with a small memory footprint (<23 MB each) is obtained running YOLOv2 inference. However, adjacent partition requires overlapping intermediate inputs resulting in 3—5× redundant computation. CoEdge [376] reduces the redundant calculation by nonoverlapping partitioning with an added data transmission cost. In another similar work, five IoT and a gateway device are employed that uses weight pruning and model partitioning using FTP [377]. An approximate 16% improvement in inference latency compared to DeepThings is demonstrated for YOLOv2 using 5 × 5 fused layer partitioning. Edgent is another distributed EI framework for a static and dynamic network architecture that exploits model partitioning and early exiting techniques [378]. The execution plan of Edgent varies depending on the available bandwidth. Incorporating raspberry pi and laptops, Edgent demonstrated low latency on-demand EI.

DistrEdge [379] is an adaptive distributed EI framework for CNN models considering the heterogeneity of edge devices, network conditions, and nonlinear computations. NVIDIA Jetson devices are employed to create a heterogenous distributed infrastructure demonstrating improved inference latency.

To facilitate distributed inference for DL models having a directed acyclic graph (DAG) structure, EdgeFlow is reported in [380]. The computation graphs are broken into a list of input requirements, a computation operator, and a forwarding table. The execution unit applies the computation operator on input to generate intermediate results. According to the forwarding table, the intermediate results are distributed to other units. Parallel executions are achieved in EdgeFlow by partitioning a DL layer into multiple independent execution units. Besides, an algorithm is developed to determine the optimal portioning and ensure the run-time efficiency. A simplified system architecture of EdgeFlow is depicted in Fig. 24. Experiments demonstrated up to a 40.2% reduction in inference latency in EdgeFlow compared to local EI while running the YOLOv5 model.

The transformer-based DL models have achieved promising results in NLP applications, such as machine translation, grammatical error correction, and abstractive summarization. For example, EdgeFormer [381] is a transformer-based encoder–decoder architecture designed for seq2seq generation considering the memory and computation constraints of edge devices. In machine translation and grammatical error correction, EdgeFormer demonstrated high parameter optimization (<10M) and faster latency (100 ms) with only a few megabytes of memory (<50-MB RAM) using two CPU cores. Another efficient and lossless acceleration of seq2seq generation is presented in [382] that exploits aggressive decoding and parallel processing. The experimental results in GPU execution using a six-layer transformer model obtained up to 9× and 5× speedup for grammatical error correction and machine translation.

XIII. USE CASES OF EDGE INFERENCE

EI has gained significant attention and is the key technology to bring AI services to the end user. Practical use cases of EI include, but are not limited to, face recognition, HAR, ADAS, speech recognition, and so on. For instance, face recognition is being actively used in door unlocking, mobile banking [383], device unlocking [384], and so on. In [385], a CNN named Mobile-FaceNets is presented that uses a global depthwise convolution filter that can perform face recognition having less than 1M parameters. MobiFace [386] is an improvement on Mobile-FaceNet using residual bottleneck [72] with an expansion layer and fast downsampling.

HAR finds significant applications in wearable sensors for fitness tracking, activity monitoring, patient monitoring, fall detection, elder care, and so on [387]. RBMs

are used in [388] for HAR applications on smartwatches. Statistical features of CNN are used in [389], which extended, in [390], adding the position features for HAR. Due to the heterogeneous sensor types and distinct user behavior, incremental learning [391] provides secondary training for the pretrained model to learn new features from incoming data. Thus, a personalized user experience with improved recognition accuracy can be achieved. In DeepSense [392], both CNN and RNN are used to tackle the inherent noisy behavior of raw activity sensor data. The CNN captures the interaction of heterogenous sensors, while the RNN captures the temporal characteristics providing an ability to differentiate significant inputs to improve predictions.

DeepEye [393] is a wearable camera capable of running five DL models on edge for image recognition tasks. Content-based image retrieval (CBIR) for edge devices has been presented in [394] that uses the low-rank approximation for feature detection and object detection resulting in a 6.1× inference speedup.

MobileDeepPill [395] is an image recognition system to identify pills for patients from pill images. The KD technique is incorporated having a large CNN model as the teacher. During inference, the student CNN is used to calculate the output class probabilities on an input pill image.

Driver fatigue or other distractions can be detected in real time using DL inference on a smartphone. In DarNet [396], driving patterns are analyzed using DL to identify distracted driving. Two sensor types are used to capture the driving patterns: the inertial measurement unit (IMU) data from the smartphone and the images captured using the IoT sensors with a camera. CNN is used to process the image data, whereas RNN processes the IMU data. Ensemble learning is used to capture the real-time distracted driving activity by combining both CNN and RNN results.

DeepEar [397] is DL-based audio sensing using a smartphone that used stacked RBM. The experiment demonstrates that sharing the same base bottom layer can sense the audio with only 6% battery consumption throughout a single day with only 3% accuracy degradation. The accuracy can be further enhanced by training shared layers and multitask learning [398].

DL-based vehicle detection to assist autonomous driving has been presented in [399], which uses the faster-RCNN network. Pruning and quantization are used together achieving a 16% reduction in model size and a 64% reduction in runtime. An RNN model is used in [400] for driving pattern analysis from raw sensor data. Parameter quantization is used to make the EI on smart automobiles with reduced memory and power consumption. Results demonstrated only 44-kB memory and 7.7-mW power consumption.

Anomaly detection of electrical equipment based on compressed AlexNet through pruning and quantization can detect the status of the instrument using images [408]. Lightweight DL models can also be used for transmission line fault detection [409] with 74.5% accuracy.

Table 10 presents a summary of the representative research works and use cases of EI on various edge devices. This table lists information from multiple viewpoints, such as used/supported DL models, adopted optimization techniques, use cases/applications, edge devices, and used performance evaluation metrics.

XIV. CHALLENGES AND OPPORTUNITIES OF EDGE INFERENCE

The DL inference on edge devices can enable highly efficient and intelligent services to the end user and minimize the traditional cloud dependence on AI services. Based on the comprehensive discussions on EI tools and techniques, this section identifies some remaining or minimally explored challenging areas and discusses the research trend to find solutions.

A. Adaptability to Data Heterogeneity

EI often requires collecting data using multiple sensors and in distinct sensing environments. For instance, if the data source changes from indoor to the street or the weather changes from cloudy to sunny, the sensor data may also get affected. Even if the environment and source of data remain the same, sensors from different manufacturers might have distinct sensing behavior. Such heterogeneity may cause inconsistency in data imposing a challenge on EI performance. Data augmentation [410], [411] and representation learning could [412] be two promising future research directions to mitigate the data inconsistency problem at the edge.

B. Automatic Mapping of DL to Hardware

The FPGA and CGRA can overcome some limitations of ASICs with the ability to reconfigure new DL operations, modules, or even network architectures. However, the conversion of trained DL models into a hardware-compatible version for inference is tedious, time-consuming, and manual. There is a significant research gap to develop tools that can automatically map the DNN models on hardware. Available tools for automatic mapping of DL models to FPGA or CGRA are less efficient. Therefore, developing efficient mapping frameworks to support DL deployment on reconfigurable devices is a challenge.

The ML frameworks help researchers quickly prototype the models for different applications. More functions and libraries should be added to these frameworks to quickly implement the compression techniques. For example, NVIDIA's SparseBLAS/cuSPARSE and Intel's MKL libraries can handle sparse matrix computations. TensorFlow provides library support for quantization during training or posttraining. Xilinx provides quantization support through the FINN-R framework for inference realization on FPGA [413]. Therefore, developing more advanced

Table 10 Summary of Representative Research on DL Inference on Edge

Reference	Deep Learning Model	Optimization Technique	Application	Edge Device	Quality Metric
[401]	MobileNet	Model selection	Image classification	NVIDIA Jetson TX2	Latency, accuracy
[314]	LSTM	Structured compression, quantization	Speech recognition	Xilinx KU060 and Virtex-7 FPGA	Energy efficiency, speed
[158]	LeNet5, VGGNet BiLSTM	Pruning	Speech, image, text recognition	Intel Edison	Memory, energy, latency
[313]	Single-layer RNN with 256 GRU	delta network update, zero-skipping	Spoken digit recognition	Xilinx Zynq-7100 FPGA	Throughput, speed, energy efficiency
[402]	7-Layer CNN	Quantization	Image classification	ARM Cortex-M	Number of operations, memory
[96]	LSTM	Pruning, structured sparsity	NLP, speech recognition	Arria-10 FPGA	Latency, energy efficiency, number of operations
[160]	LSTM	Pruning, quantization	Speech recognition	XCKU060 FPGA	Number of operations, memory
[164]	CNN	Quantization, optimized convolution	Image classification	Ultra96 SoC	Accuracy, inference speed
[200]	AlexNet VGGNet	Sparsification	Speech, image recognition	Qualcomm Snapdragon, 400Nvidia Tegra K1, ARM Cortex M0 and M3	Memory
[140]	Binary CNN	Binarization, optimized convolution	Image classification	CPU	Accuracy, inference speed
[161]	LeNet, AlexNet, VGGNet	Model selection	Audio, image, activity classification	Wearables, smartphones, development boards, smart home devices	Memory, energy, latency
[403]	LSTM	Model selection, software optimization	Blood glucose prediction	ARM Cortex-M4	Inference time, power consumption
[162]	YOLO, MatConvNet	GPU	Object detection	Samsung Galaxy S7	Latency, caching hit rate
[166]	FCN, CNN	Pruning, MAC optimization, approximate memory	Digit and object recognition	RTL	Accuracy, energy efficiency
[375]	CNN	Parallel computing	Object detection	Raspberry Pi model 3 (ARM v8)	Memory saving, inference speed, throughput
[259]	24-layer CNN, LSTM	Heterogenous processor	Image and hand gesture classification	Nexus 6, Nexus 5x	Latency
[404]	CNN, AlexNet, VGG 16	Zero-skipping, weight pruning	Image classification	ZYNQ XC7020 and ZYNQ XC7Z7045	Inference time
[405]	CNN, VGG 16, ResNet-50	Filter pruning	Image classification, object detection, segmentation	GPU, CPU, mobile phone	Speed up, memory saving, generalization
[406]	CNN, AlexNet, MLP	Binarization	Classification, digit recognition,	Stratix-V FPGA	Speed, inference time, energy efficiency
[407]	AlexNet, ResNet-18, ResNet-50	Optimized multiplication, quantization	Image classification	Xilinx Virtex 5/6 FPGAs, UltraScale/UltraScale+ FPGAs	Speedups and power savings
[284]	AlexNet	Heterogenous processor	Speech, image recognition	Qualcomm Snapdragon, 800 Nvidia Tegra K1	Memory, energy

libraries and frameworks to support training and posttraining compression of DL models is imperative.

C. Developing Benchmarks

Proper benchmark standards to assess the performance of EI are significantly important. EI requires a universal and comprehensive set of quality metrics to execute meaningful comparisons of DL models, optimization algorithms, and hardware platforms.

Benchmark datasets and models are limited. Most compression techniques and hardware performance are evaluated on CNN (e.g., AlexNet and VGG16) on the ImageNet dataset. Therefore, more benchmark datasets and DL models need to be developed for other applications and input data types. Due to the error-accumulating nature of DL models, the optimization performance of large DL models (e.g., layers > 50) should also be studied. Besides, extending the compression algorithms to be universal for both compact and deep networks is challenging.

D. Automatic, Joint, and Edge Aware Compression

Most existing compression techniques require manual intervention and expert inputs (e.g., quantization bit width, rank value in decomposition, and layerwise sparsity). Compression hyperparameters, such as pruning ratio, quantization bit widths, pruning pattern for different layers, and epochs, need to be automatically adjustable for optimum memory usage and minimum computations with acceptable accuracy degradation. Therefore, developing an automatic compression technique can be an interesting research domain.

Most of the model compression algorithms are employed posttraining. Due to the iterative nature of the training

process, implementing compression during training is infeasible and complicated. During training, joint automated exploration of compression algorithms can increase the compression and improve hardware realization. Thus, realizing the compression for faster convergence and reduced computation needs can be an exciting research area to explore.

There are lots of DNN compression techniques being continuously evolving. However, fair comparison metrics are lacking. Therefore, there are research opportunities to develop compression evaluations metric to identify the performance of the compression techniques.

E. Algorithm–Hardware Codesign

The lightweight DL model and compression techniques should be developed considering the hardware architecture. Therefore, hardware-algorithm codesign can produce more efficient EI. Model compression needs to be associated with architecture optimization for the highest efficiency. For instance, different edge hardware platforms have varying capabilities for handling different precisions. Often accelerators support a uniform bit-width tensor. In such cases, handling distinct bit-width precision needs zero padding incurring inefficient memory usage and processing. In such cases, hardware-aware compression can improve processing efficiency.

Frameworks such as DNNWeaver [414], DNNBuilder [415], T2S-Tensor [416], and HeteroCL [417] have been developed recently for FPGA, ASIC, and other DL hardware accelerators for efficient EI. However, most frameworks still support the fixed bit-width tensors and cannot handle the sparsity caused by model compression. Therefore, reconfigurable architectures need to be developed for rapid prototyping and evaluation allowing sparsity.

F. Neural Architecture Search for Edge Inference

NAS could obtain efficient models with high accuracy. The limitations are the computational load and time required for the NAS algorithms that are high. Thus, there are research opportunities to considerably improve the NAS algorithms to obtain highly compact network architectures targeting edge. Due to the vast search space for NAS, notable efforts are required for the automated exploration of model-accelerator codesigns.

Different edge hardware platforms or neural accelerators have distinct properties and processing capabilities. Therefore, a hardware-aware NAS algorithm can consider the performance feedback from edge hardware to generate the optimum neural architecture for specific hardware. For instance, ProxylessNAS [418] can find neural architecture to fit the hardware. However, due to the availability of numerous hardware devices (e.g., IoT devices), there is a proportional growth of search space and increased complexity of NAS.

G. Developing Flexible ASICs

One of the main limitations of ASIC is not being adaptable to changes resulting in performance inefficiency. There are significant research opportunities to fill the gap of adding some flexibility in ASICs and SoCs to cope and adapt to the improved DL operations and algorithms. If some flexibility can be added, this will improve the silicon efficiency and can be cost-friendly. Thus, dynamic reconfigurability is expected to be rooted in the ASIC or SoCs. Due to extra logic and wiring, minimizing the power consumption and area of such ASIC could be an active area of research.

H. Neural Accelerators to Handle Sparsity

General-purpose processors, such as CPU, GPU, or even FPGA, are not specialized to handle the sparse, irregular tensors, and low-precision computation resulting from different compression techniques. Hence, special architecture should be developed to accelerate the compressed DNN models. In addition, combining multiple compression techniques for optimum hardware performance is a promising research direction.

General hardware platforms for EI can process regular tensors and data flow. However, compression techniques result in many irregularities in tensor processing. For instance, bit-adaptive computing and high reconfigurability are required for sparse tensors with flexible bit width. Hardware accelerators cannot inherently compute sparse tensors. Data such as activations, parameters, and gradients include zero values that require to be retrieved from memory to the PEs. The zero computations are not automatically skipped requiring PEs' execution time on the ineffective zero computations. In unstructured sparsity, nonzero values are scattered randomly across the tensor resulting in irregular processing. Therefore, to handle the significant sparsity in modern DL algorithms, additional techniques are necessary to store, process, and compute the nonzero elements.

I. Focusing on Other DL Techniques

Model compression techniques reduce the computation and memory needs of edge devices to run efficient inferences. However, many optimization techniques, such as pruning and batching, targeted the dense layers. In modern architectures, few dense layers are used, so these techniques become less effective in terms of overall network performance. The transition from research prototypes to industry-ready products is one of the major challenges. Many of the techniques developed focus only on the FNN and CNN due to a wide variety of applications. However, the technology is moving faster with the wide acceptance of other DL methods. For example, sequence analysis is prominent in commercial edge devices for voice recognition, NLP, and many more. Therefore, the focus should be given to the understanding of how existing techniques

work on time series or sequence analysis on edge and the development of new techniques.

J. Training on the Edge

Training DL models are performed mostly in the cloud. Such training is isolated and static, having no subsequent attempts to retain the learned knowledge for the future. Besides requiring large datasets, the application domain is also limited. Adding the retraining capability and transfer learning on edge can open various services where data are not static, and continuous learning is necessary from new data. Though accelerators such as ScaleDeep [419] and HyPar [420] are proposed for training on edge, these architectures do not consider the sparsity, variable precision of gradients, parameters, and activations. Hence, there are significant improvement opportunities to leverage layer-level optimization, boost performance, and improve energy efficiency by supporting mixed precision. As training is more computation-intensive, challenges are severe for edge training compared to EI.

Most of the edge training techniques are developed adopting supervised learning assuming the availability of sufficient labeled data of good quality. These assumptions pose a serious challenge to these algorithms to deal with real-life data, which are mostly unlabeled and sparse. Active learning [421] can deal with the unlabeled data problem up to a certain extent. However, active learning works well for small datasets but requires manual intervention for annotation. Federated learning could be another solution through decentralized training. Instead of uploading and making a centralized database, federated learning is collaboratively trained on multiple local nodes using local datasets, and learned parameters are exchanged. Using the parameters from multiple nodes a global model is generated and shared by all local nodes. Other possible solutions to data scarcity at the edge might consider incremental learning and data augmentation.

Lifelong ML (LML) [422] could bring human-like intelligence to edge devices and could be the possible future research direction to tackle the environmental dynamics and scarcity of data. LML is a continuous learning paradigm that accumulates knowledge from the past and adapts it to the present. However, the concept of LML is developed considering the high computational capacity of the machines and is not targeting edge devices. Therefore, making LML applicable for edge training and inference is one of the challenges and could bring many research opportunities.

K. Increased Demand of Communication Resources

The increase in communication resources' demand in data collection, cloud transfer, edge deployment of large models, and distributed training and inference are a few communication challenges associated with EI. The DL models are data-driven, which require edge big-data transfer from edge to the cloud for storage and training. In addition, even after significant compression and algorithm–hardware optimization, the model size can be large. Transferring such a large model to deploy on edge requires connectivity. Different communication strategies, such as wired connection, cellular networks, and opportunistic spectrum, have been explored. Data collection edge devices require interconnection with the edge clusters for proper data transfer. This data transfer from edge devices consumes spectral resources and puts an additional burden on the hardware costs. In federated learning, an established connection between the cloud server and edge devices is critical to upload the updates from edge nodes and download the accumulated updates from shared local models. Minimizing the number of update rounds, efficient spectral usage, and cost-effective communication are a few challenges of federated learning. Multiaccess edge computing (MEC) is an emerging technology [423] that has the potential to ensure optimal coordination between the spectral, storage, and computing resources within the limited power and latency budget. By sharing the communication resources in proximity, MEC has the potential to meet the spectral demand in data collection and distributed training and inference. In addition, the envisioned 6G network infrastructure will eliminate several communication resources burden through high-capacity satellites and possible exploitation of terahertz and optical frequency bands [20].

L. Explainability in Edge Inference

Advances in AI facilitate highly accurate decision-making in numerous application domains. However, there is an increasing concern about transparency, ethics, and accountability for the widespread adoption of AI in healthcare, jurisprudence, and law enforcement. The rapid emergence of DL makes it tough to justify and explain the outcomes. Thus, explainable AI (XAI) brings a novel paradigm shift to provide transparency in DL algorithms [424]. The ever-increasing intelligent edge devices incorporating DL inference impose a tremendous challenge in ensuring that the output is harmless. Due to multiple optimizations, often, the accuracy is compromised in EI. Therefore, incorporating explainability remains a minimally explored and challenging research direction.

XV. CONCLUSION

Most DL model designs are concerned with empirically attaining the highest achievable accuracy. However, leveraging the full potential of DL through embedded AI capability for the end-user application requires adaptability and applicability to various domains and future evolution. This article analyzes some avenues that preceding works have accomplished to optimize DL inference processing on edge. For example, edge-oriented compression techniques and available EI hardware platforms are presented. Different edge-specific quality metrics, such as model size, accuracy,

latency, throughput, and cost, are analyzed. Memory access techniques and data flow often dominate power consumption. Thus, techniques developed to optimize memory access are also described. In addition, available software packages and ML frameworks for algorithm–hardware codesign techniques for training, optimizing, and deploying DL for EI are included. In summary, this article presents a comprehensive review and systematic classification of EI research and a thorough exploration of each component. Moreover, open research questions, challenges, and possible future directions are discussed. This article will help researchers cope with the current state of the tools and techniques for EI and develop next-generation AI-enabled end-user systems. ■

REFERENCES

[1] Y. Bengio, I. Goodfellow, and A. Courville, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2016.

[2] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[3] S. Wendelken and C. MacGillivray. *Worldwide and U.S. IoT Cellular Connections Forecast, 2021–2025*. Accessed: Feb. 17, 2022. [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=US47296121

[4] S. Pouyanfar et al., "A survey on deep learning: Algorithms, techniques, and applications," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 1–36, 2018.

[5] M. Z. Alom et al., "A state-of-the-art survey on deep learning theory and architectures," *Electronics*, vol. 8, no. 3, p. 292, Mar. 2019.

[6] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "Model compression and acceleration for deep neural networks: The principles, progress, and challenges," *IEEE Signal Process. Mag.*, vol. 35, no. 1, pp. 126–136, Jan. 2018.

[7] E. Wang et al., "Deep neural network approximation for custom hardware: Where we've been, where we're going," *ACM Comput. Surv.*, vol. 52, no. 2, p. 40, 2019.

[8] M. Capra, B. Bussolino, A. Marchisio, M. Shafique, G. Masera, and M. Martina, "An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks," *Future Internet*, vol. 12, no. 7, p. 113, Jul. 2020.

[9] D. Moolchandani, A. Kumar, and S. R. Sarangi, "Accelerating CNN inference on ASICs: A survey," *J. Syst. Archit.*, vol. 113, Feb. 2021, Art. no. 101887.

[10] B. L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proc. IEEE*, vol. 108, no. 4, pp. 485–532, Apr. 2020.

[11] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019.

[12] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks," *Neural Comput. Appl.*, vol. 32, pp. 1–31, Oct. 2018.

[13] M. Li et al., "The deep learning compiler: A comprehensive survey," 2020, *arXiv:2002.03794*.

[14] A. Ignatov et al., "AI benchmark: All about deep learning on smartphones in 2019," 2019, *arXiv:1910.06663*.

[15] S. Dave, R. Baghdadi, T. Nowatzki, S. Avancha, A. Shrivastava, and B. Li, "Hardware acceleration of sparse and irregular tensor computations of ML models: A survey and insights," *Proc. IEEE*, vol. 109, no. 10, pp. 1706–1752, Oct. 2021.

[16] M. P. Véstias, R. P. Duarte, J. T. de Sousa, and H. C. Neto, "Moving deep learning to the edge," *Algorithms*, vol. 13, no. 5, p. 125, May 2020.

[17] R. Wu, X. Guo, J. Du, and J. Li, "Accelerating neural network inference on FPGA-based platforms—A survey," *Electronics*, vol. 10, no. 9, p. 1025, Apr. 2021.

[18] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proc. IEEE*, vol. 107, no. 8, pp. 1738–1762, Aug. 2019.

[19] J. Chen and X. Ran, "Deep learning with edge computing: A review," *Proc. IEEE*, vol. 107, no. 8, pp. 1655–1674, Aug. 2019.

[20] D. Xu et al., "Edge intelligence: Empowering intelligence to the edge of network," *Proc. IEEE*, vol. 109, no. 11, pp. 1778–1837, Nov. 2021.

[21] U. Drolia, K. Guo, J. Tan, R. Gandhi, and P. Narasimhan, "Cachier: Edge-caching for recognition applications," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 276–286.

[22] U. Drolia, K. Guo, and P. Narasimhan, "Precog: Prefetching for image recognition applications at the edge," in *Proc. Symp. Edge Comput.*, 2017, p. 17.

[23] P. Guo, B. Hu, R. Li, and W. Hu, "FoggyCache: Cross-device approximate computation reuse," in *Proc. 24th Annu. Int. Conf. Mobile Comput. Netw.*, Oct. 2018, pp. 19–34.

[24] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu, "DeepCache: Principled cache for mobile deep vision," 2017, *arXiv:1712.01670*.

[25] Y. Kang et al., "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Comput. Archit. News*, vol. 52, no. 4, pp. 615–629, 2017.

[26] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon, "IONN: Incremental offloading of neural network computations from mobile devices to edge servers," in *Proc. ACM Symp. Cloud Comput.*, Oct. 2018, pp. 401–411.

[27] M. M. H. Shuvo et al., "Machine learning embedded smartphone application for early-stage diabetes risk assessment," in *Proc. IEEE Int. Symp. Med. Meas. Appl. (MeMeA)*, Jun. 2022, pp. 1–6.

[28] S. Wang et al., "When edge meets learning: Adaptive control for resource-constrained distributed machine learning," in *Proc. IEEE Conf. Comput. Commun.*, Apr. 2018, pp. 63–71.

[29] C. Gupta et al., "ProtoNN: Compressed and accurate KNN for resource-scarce devices," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 1331–1340.

[30] D. K. Dennis et al., "EdgeML: Machine learning for resource-constrained edge devices," version 0.4, 2020. [Online]. Available: https://github.com/Microsoft/EdgeML

[31] A. Kumar, S. Goyal, and M. Varma, "Resource-efficient machine learning in 2 KB RAM for the Internet of Things," in *Proc. 34th Int. Conf. Mach. Learn.*, vol. 70, Sep. 2017, pp. 1935–1944. [Online]. Available: https://proceedings.mlr.press/v70/kumar17a.html

[32] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. Cambridge, MA, USA: MIT Press, 2012.

[33] P. Refaeilzadeh, L. Tang, H. Liu, L. Angeles, and C. D. Scientist, "Cross-validation," *Encyclopedia Database Syst.*, vol. 5, pp. 532–538, Jan. 2020.

[34] T.-T. Wong, "Performance evaluation of classification algorithms by k-fold and leave-one-out cross validation," *Pattern Recognit.*, vol. 48, no. 9, pp. 2839–2846, 2015.

[35] S. Ruder, "An overview of gradient descent optimization algorithms," 2016, *arXiv:1609.04747*.

[36] P. Cunningham, M. Cord, and S. J. Delany, "Supervised learning," in *Machine Learning Techniques for Multimedia*. Berlin, Germany: Springer, 2008, pp. 21–49.

[37] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms," in *Proc. 23rd Int. Conf. Mach. Learn.*, 2006, pp. 161–168.

[38] T. Hastie, R. Tibshirani, and J. Friedman, "Unsupervised learning," in *The Elements of Statistical Learning*. Springer, 2009, pp. 485–585.

[39] J. E. Van Engelen and H. H. Hoos, "A survey on semi-supervised learning," *Mach. Learn.*, vol. 109, no. 2, pp. 373–440, Feb. 2020.

[40] X. J. Zhu, "Semi-supervised learning literature survey," Dept. Comput. Sci., Univ. Wisconsin-Madison, Madison, Wisconsin, 2005.

[41] Y. Li, "Deep reinforcement learning," *CoRR*, vol. abs/1810.06339, 2018. [Online]. Available: http://arxiv.org/abs/1810.06339.

[42] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *J. Artif. Intell. Res.*, vol. 4, no. 1, pp. 237–285, Jan. 1996.

[43] M. Wiering and M. van Otterlo, *Reinforcement Learning*, vol. 12. Berlin, Germany: Springer, 2012.

[44] F. Agostinelli, M. Hoffman, P. Sadowski, and P. Baldi, "Learning activation functions to improve deep neural networks," 2014, *arXiv:1412.6830*.

[45] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[46] T. B. Brown et al., "Language models are few-shot learners," 2020, *arXiv:2005.14165*.

[47] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," 2015, *arXiv:1506.02626*.

[48] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, Jan. 2014.

[49] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[50] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, vol. 25, Dec. 2012, pp. 1097–1105.

[51] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.

[52] C. Szegedy et al., "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2015, pp. 1–9.

[53] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[54] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 2818–2826.

[55] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-ResNet and the impact of residual connections on learning," in *Proc. Conf. Artif. Intell.*, 2017, pp. 1–7.

[56] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 1251–1258.

[57] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 4700–4708.

[58] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018,

pp. 7132–7141.
[59] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," 2017, *arXiv:1707.07012*.
[60] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *J. Mach. Learn. Res.*, vol. 20, no. 1, pp. 1997–2017, 2019.
[61] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 2016, *arXiv:1611.01578*.
[62] *AutoML*. Accessed: Feb. 4, 2022. [Online]. Available: https://cloud.google.com/automl/
[63] *AutoKeras*. Accessed: Feb. 11, 2022. [Online]. Available: https://autokeras.com/
[64] A. Howard et al., "Searching for MobileNetV3," 2019, *arXiv:1905.02244*.
[65] T. Yang et al., "NetAdapt: Platform-aware neural network adaptation for mobile applications," 2018, *arXiv:1804.03230*.
[66] C. Liu et al., "Progressive neural architecture search," in *Proc. Eur. Conf. Comput. Vis.*, 2018, pp. 19–34.
[67] M. Tan et al., "MnasNet: Platform-aware neural architecture search for mobile," in *Proc. Conf. Comput. Vis. Pattern Recognit.*, Jun. 2019, pp. 2815–2823.
[68] H. Wang et al., "HAT: Hardware-aware transformers for efficient natural language processing," 2020, *arXiv:2005.14187*.
[69] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," 2018, *arXiv:1806.09055*.
[70] A. G. Howard et al., "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*.
[71] M. Maaz et al., "EdgeNeXt: Efficiently amalgamated CNN-transformer architecture for mobile vision applications," 2022, *arXiv:2206.10589*.
[72] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 4510–4520.
[73] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size," 2016, *arXiv:1602.07360*.
[74] A. Gholami et al., "SqueezeNext: Hardware-aware neural network design," 2018, *arXiv:1803.10615*.
[75] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An extremely efficient convolutional neural network for mobile devices," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 6848–6856.
[76] G. Huang, S. Liu, L. V. D. Maaten, and K. Q. Weinberger, "CondenseNet: An efficient DenseNet using learned group convolutions," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 2752–2761.
[77] Y. Xiong, H. J. Kim, and V. Hedau, "ANTNets: Mobile convolutional neural networks for resource efficient image classification," 2019, *arXiv:1904.03775*.
[78] S. Winograd, *Arithmetic Complexity of Computations*, vol. 33. Philadelphia, PA, USA: SIAM, 1980.
[79] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 4013–4021.
[80] L. Meng and J. Brothers, "Efficient Winograd convolution via integer arithmetic," 2019, *arXiv:1901.01965*.
[81] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *Proc. IEEE 25th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2017, pp. 101–108.
[82] S. Kala, J. Mathew, B. R. Jose, and S. Nalesh, "UniWiG: Unified winograd-GEMM architecture for accelerating CNN on FPGAs," in *Proc. 32nd Int. Conf. VLSI Design 18th Int. Conf. Embedded Syst. (VLSID)*, Jan. 2019, pp. 209–214.
[83] M. Hardieck, M. Kumm, K. Möller, and P. Zipf, "Reconfigurable convolutional kernels for neural networks on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2019, pp. 43–52.
[84] R. Jozefowicz, W. Zaremba, and I. Sutskever, "An empirical exploration of recurrent network architectures," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 2342–2350.
[85] Z. Wu and S. King, "Investigating gated recurrent networks for speech synthesis," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Mar. 2016, pp. 5140–5144.
[86] J. van der Westhuizen and J. Lasenby, "The unreasonable effectiveness of the forget gate," 2018, *arXiv:1804.04849*.
[87] G.-B. Zhou, J. Wu, C.-L. Zhang, and Z.-H. Zhou, "Minimal gated unit for recurrent neural networks," *Int. J. Automat. Comput.*, vol. 13, no. 3, pp. 226–234, 2016.
[88] D. Neil, M. Pfeiffer, and S.-C. Liu, "Phased LSTM: Accelerating recurrent network training for long or event-based sequences," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 3882–3890.
[89] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition," 2014, *arXiv:1402.1128*.
[90] O. Kuchaiev and B. Ginsburg, "Factorization tricks for LSTM networks," 2017, *arXiv:1703.10722*.
[91] Y. Wu et al., "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016, *arXiv:1609.08144*.
[92] S. Zhang et al., "Architectural complexity measures of recurrent neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 1822–1830.
[93] N. Kalchbrenner, I. Danihelka, and A. Graves, "Grid long short-term memory," 2015, *arXiv:1507.01526*.
[94] Z. He, S. Gao, L. Xiao, D. Liu, H. He, and D. Barber, "Wider and deeper, cheaper and faster: Tensorized LSTMs for sequence learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 1–11.
[95] X. Dai, H. Yin, and N. K. Jha, "NeST: A neural network synthesis tool based on a grow-and-prune paradigm," *IEEE Trans. Comput.*, vol. 68, no. 10, pp. 1487–1497, Oct. 2019.
[96] S. Cao et al., "Efficient and effective sparse LSTM on FPGA with bank-balanced sparsity," in *Proc. Int. Symp. Field-Prog. Gate Arrays*, 2019, pp. 63–72.
[97] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern GPUs," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 359–371.
[98] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 29, 2016, pp. 2074–2082.
[99] H. Mao et al., "Exploring the regularity of sparse structure in convolutional neural networks," 2017, *arXiv:1705.08922*.
[100] Q. Huang, K. Zhou, S. You, and U. Neumann, "Learning to prune filters in convolutional neural networks," in *Proc. IEEE Winter Conf. Appl. Comput. Vis. (WACV)*, Mar. 2018, pp. 709–718.
[101] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 548–560, 2017.
[102] Z. Jelčicová and M. Verhelst, "Delta keyword transformer: Bringing transformers to the edge through dynamically pruned multi-head self-attention," 2022, *arXiv:2204.03479*.
[103] S. Huang et al., "An automatic and efficient BERT pruning for edge AI systems," in *Proc. Int. Symp. Quality Electron. Design*, Jun. 2022, pp. 1–6.
[104] F. Manessi, A. Rozza, S. Bianco, P. Napoletano, and R. Schettini, "Automated pruning for deep neural network compression," in *Proc. 24th Int. Conf. Pattern Recognit. (ICPR)*, Aug. 2018, pp. 657–664.
[105] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," 2016, *arXiv:1611.06440*.
[106] Z. You, K. Yan, J. Ye, M. Ma, and P. Wang, "Gate decorator: Global filter pruning method for accelerating deep convolutional neural networks," 2019, *arXiv:1909.08174*.
[107] H. Wang, Q. Zhang, Y. Wang, L. Yu, and H. Hu, "Structured pruning for efficient ConvNets via incremental regularization," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2019, pp. 1–8.
[108] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 5687–5695.
[109] R. J. R. Struharik, B. Z. Vukobratović, A. M. Erdeljan, and D. M. Rakanović, "CoNNa–Hardware accelerator for compressed convolutional neural networks," *Microprocessors Microsyst.*, vol. 73, Mar. 2020, Art. no. 102991.
[110] E. Nurvitadhi et al., "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?" in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2017, pp. 5–14.
[111] Y. Shen, M. Ferdman, and P. Milder, "Escher: A CNN accelerator with flexible buffering to minimize off-chip transfer," in *Proc. IEEE 25th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2017, pp. 93–100.
[112] P. Micikevicius et al., "Mixed precision training," 2017, *arXiv:1710.03740*.
[113] N. Wang, J. Choi, D. Brand, C. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," 2018, *arXiv:1812.08011*.
[114] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," in *Proc. Int. Conf. Learn. Represent.*, 2016, pp. 1–15.
[115] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *Proc. 33rd Int. Conf. Int. Conf. Mach. Learn.*, vol. 48, Jun. 2016, pp. 2849–2858.
[116] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proc. 32nd Int. Conf. Mach. Learn. (ICML)*, vol. 37, 2015, pp. 1737–1746.
[117] S. Anwar, K. Hwang, and W. Sung, "Fixed point optimization of deep convolutional neural networks for object recognition," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Apr. 2015, pp. 1131–1135.
[118] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2014, pp. 10–14.
[119] B. Moons and M. Verhelst, "A 0.3–2.6 TOPS/W precision-scalable processor for real-time large-scale ConvNets," in *Proc. IEEE Symp. VLSI Circuits (VLSI-Circuits)*, Jun. 2016, pp. 1–2.
[120] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on CPUs," in *Proc. Deep Learn. Unsupervised Feature Learn. Workshop*, 2011, pp. 1–8.
[121] M. A. Nasution, D. Chahyati, and M. I. Fanany, "Faster R-CNN with structured sparsity learning and Ristretto for mobile environment," in *Proc. Int. Conf. Adv. Comput. Sci. Inf. Syst. (ICACSIS)*, Oct. 2017, pp. 309–314.
[122] S. H. F. Langroudi, T. Pandit, and D. Kudithipudi, "Deep learning inference on embedded devices: Fixed-point vs posit," in *Proc. 1st Workshop Energy Efficient Mach. Learn. Cognit. Comput. Embedded Appl. (EMC2)*, Mar. 2018, pp. 19–23.
[123] P. Judd, J. Albericio, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," *IEEE Comput. Archit. Lett.*, vol. 16, no. 1, pp. 80–83, Aug. 2017.
[124] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "UNPU: A 50.6 TOPS/W unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2018, pp. 218–220.

[125] S. Sharify, A. D. Lascorz, K. Siu, P. Judd, and A. Moshovos, "Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, Jun. 2018, pp. 1–6.

[126] H. Sharma et al., "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 764–775.

[127] S. Ryu, H. Kim, W. Yi, and J.-J. Kim, "BitBlade: Area and energy-efficient precision-scalable neural network accelerator with bitwise summation," in *Proc. 56th Annu. Design Autom. Conf.*, Jun. 2019, pp. 1–6.

[128] J. Wang, Q. Lou, X. Zhang, C. Zhu, Y. Lin, and D. Chen, "Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA," in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2018, pp. 163–1636.

[129] A. D. Lascorz et al., "ShapeShifter: Enabling fine-grain datawidth adaptation in deep learning," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 28–41.

[130] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "YodaNN: An ultra-low power convolutional neural network accelerator based on binary weights," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2016, pp. 236–241.

[131] K. Ando et al., "BRein memory: A 13-layer 4.2 K neuron/0.8 M synapse binary/ternary reconfigurable in-memory deep neural network accelerator in 65 nm CMOS," in *Proc. Symp. VLSI Circuits*, Jun. 2017, pp. C24–C25.

[132] M. Courbariaux, Y. Bengio, and J. P. David, "BinaryConnect: Training deep neural networks with binary weights during propagations," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 3123–3131.

[133] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," 2015, *arXiv:1510.03009*.

[134] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to $+1$ or -1," 2016, *arXiv:1602.02830*.

[135] D. Soudry, I. Hubara, and R. Meir, "Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights," in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 963–971.

[136] S. K. Esser, R. Appuswamy, P. Merolla, J. V. Arthur, and D. S. Modha, "Backpropagation for energy-efficient neuromorphic computing," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 1117–1125.

[137] M. Courbariaux and Y. Bengio, "BinaryNet: Training deep neural networks with weights and activations constrained to $+1$ or -1," 2016, *arXiv:1602.02830*.

[138] S. Liu and H. Zhu, "Binary convolutional neural network with high accuracy and compression rate," in *Proc. 2nd Int. Conf. Algorithms, Comput. Artif. Intell.*, Dec. 2019, pp. 43–48.

[139] Z. Liu et al., "Bi-Real net: Enhancing the performance of 1-bit CNNs with improved representational capability and advanced training algorithm," in *Proc. Eur. Conf. Comput. Vis.*, 2018, pp. 722–737.

[140] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNORNet: ImageNet classification using binary convolutional neural networks," in *Proc. Eur. Conf. Comput. Vis.*, 2016, pp. 525–542.

[141] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Proc. Advances in Neural Information Processing Systems*, vols. 4107–4115. Red Hook, NY, USA: Curran Associates, 2016.

[142] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," 2016, *arXiv:1609.07061*.

[143] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients," 2016, *arXiv:1606.06160*.

[144] Z. Cai, X. He, J. Sun, and N. Vasconcelos, "Deep learning with low precision by half-wave Gaussian quantization," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 5918–5926.

[145] F. Li, B. Liu, X. Wang, B. Zhang, and J. Yan, "Ternary weight networks," 2016, *arXiv:1605.04711*.

[146] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," 2016, *arXiv:1612.01064*.

[147] S. K. Esser et al., "Convolutional networks for fast, energy-efficient neuromorphic computing," in *Proc. Nat. Acad. Sci. USA*, vol. 113, no. 41, pp. 11441–11446, 2016.

[148] M. Covell, D. Marwood, S. Baluja, and N. Johnston, "Table-based neural units: Fully quantizing networks for multiply-free inference," 2019, *arXiv:1906.04798*.

[149] W. Chen, J. T. Wilson, S. Tyree, K. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," in *Proc. ICML*, 2015, pp. 2285–2294.

[150] E. H. Lee, D. Miyashita, E. Chai, B. Murmann, and S. S. Wong, "LogNet: Energy-efficient neural networks using logarithmic computation," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Mar. 2017, pp. 5900–5904.

[151] M. M. H. Shuvo, O. Hassan, D. Parvin, M. Chen, and S. K. Islam, "An optimized hardware implementation of deep learning inference for diabetes prediction," in *Proc. IEEE Int. Instrum. Meas. Technol. Conf. (I2MTC)*, May 2021, pp. 1–6.

[152] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless CNNs with low-precision weights," 2017, *arXiv:1702.03044*.

[153] C. Xu et al., "Alternating multi-bit quantization for recurrent neural networks," 2018, *arXiv:1802.00150*.

[154] P. Wang, X. Xie, L. Deng, G. Li, D. Wang, and Y. Xie, "HitNet: Hybrid ternary recurrent neural network," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 601–611.

[155] A. Bie, B. Venkitesh, J. Monteiro, M. A. Haidar, and M. Rezagholizadeh, "A simplified fully quantized transformer for end-to-end speech recognition," 2019, *arXiv:1911.03604*.

[156] Z. Li and Q. Gu, "I-ViT: Integer-only quantization for efficient vision transformer inference," 2022, *arXiv:2207.01405*.

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

[158] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "DeepIoT: Compressing deep neural network structures for sensing systems with a compressor-critic framework," in *Proc. 15th ACM Conf. Embedded Netw. Sensor Syst.*, 2017, pp. 1–4.

[159] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient neural network kernels for arm Cortex-M CPUs," 2018, *arXiv:1801.06601*.

[160] S. Han et al., "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2017, pp. 75–84.

[161] S. Liu et al., "AdaDeep: A usage-driven, automated deep model compression framework for enabling ubiquitous intelligent mobiles," *IEEE Trans. Mobile Comput.*, vol. 20, no. 12, pp. 3282–3297, Dec. 2020.

[162] L. N. Huynh, Y. Lee, and R. K. Balan, "DeepMon: Mobile GPU-based deep learning framework for continuous vision applications," in *Proc. 15th Annu. Int. Conf. Mobile Syst., Appl., Services*, Jun. 2017, pp. 82–95.

[163] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," 2014, *arXiv:1412.6115*.

[164] Y. Yang et al., "Synetgy: Algorithm-hardware co-design for ConvNet accelerators on embedded FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Prog. Gate Arrays*, 2019, pp. 23–32.

[165] S. Baluja, D. Marwood, M. Covell, and N. Johnston, "No multiplication? No floating point? No problem! Training networks for efficient inference," 2018, *arXiv:1809.09244*.

[166] S. S. Sarwar, S. Venkataramani, A. Raghunathan, and K. Roy, "Multiplier-less artificial neurons exploiting error resiliency for energy-efficient neural computing," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, 2016, pp. 145–150.

[167] S. S. Sarwar et al., "Energy efficient neural computing: A study of cross-layer approximations," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 8, no. 4, pp. 796–809, Dec. 2018.

[168] A. Ranjan, A. Raha, V. Raghunathan, and A. Raghunathan, "Approximate memory compression for energy-efficiency," in *Proc. IEEE/ACM Int. Symp. Low Power Electron. Design (ISLPED)*, Jul. 2017, pp. 1–6.

[169] F. Sampaio, M. Shafique, B. Zatt, S. Bampi, and J. Henkel, "Approximation-aware multi-level cells STT-RAM cache architecture," in *Proc. Int. Conf. Compil., Archit. Synth. Embedded Syst. (CASES)*, Oct. 2015, pp. 79–88.

[170] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu, "ApproxANN: An approximate computing framework for artificial neural network," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, 2015, pp. 701–706.

[171] Y. Fan, X. Wu, J. Dong, and Z. Qi, "AxDNN: Towards the cross-layer design of approximate DNNs," in *Proc. 24th Asia South Pacific Design Autom. Conf.*, Jan. 2019, pp. 317–322.

[172] X. Xu, S. Das, and K. Kreutz-Delgado, "ApproxDBN: Approximate computing for discriminative deep belief networks," 2017, *arXiv:1704.03993*.

[173] X. He, L. Ke, W. Lu, G. Yan, and X. Zhang, "AxTrain: Hardware-oriented neural network training for approximate inference," in *Proc. Int. Symp. Low Power Electron. Design*, Jul. 2018, pp. 1–6.

[174] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "AxNN: Energy-efficient neuromorphic systems using approximate computing," in *Proc. Int. Symp. Low Power Electron. Design*, Aug. 2014, pp. 27–32.

[175] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," 2015, *arXiv:1511.06530*.

[176] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up convolutional neural networks using fine-tuned CP-decomposition," 2014, *arXiv:1412.6553*.

[177] A. Ren et al., "SC-DCNN: Highly-scalable deep convolutional neural network using stochastic computing," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 405–418, 2017.

[178] L. Wang and K.-J. Yoon, "Knowledge distillation and student-teacher learning for visual intelligence: A review and new outlooks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 6, pp. 3048–3068, Jun. 2022.

[179] H. Sim and J. Lee, "Log-quantized stochastic computing for memory and computation efficient DNNs," in *Proc. 24th Asia South Pacific Design Autom. Conf.*, Jan. 2019, pp. 280–285.

[180] R. Hojabr et al., "SkippyNN: An embedded stochastic-computing accelerator for convolutional neural networks," in *Proc. 56th ACM/IEEE Design Automat. Conf. (DAC)*, Jun. 2019, pp. 1–6.

[181] C. Bucilu, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2006, pp. 535–541.

[182] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015, *arXiv:1503.02531*.

[183] B. Heo, M. Lee, S. Yun, and J. Y. Choi, "Knowledge transfer via distillation of activation boundaries formed by hidden neurons," in *Proc. AAAI Conf. Artif. Intell. (AAAI)*, 2019, vol. 33, no. 1, pp. 3779–3787.

[184] Z. Huang and N. Wang, "Like what you like: Knowledge distill via neuron selectivity transfer," 2017, *arXiv:1707.01219*.

[185] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, "FitNets: Hints for thin deep nets," 2014, *arXiv:1412.6550*.

[186] D. Li, X. Wang, and D. Kong, "DeepRebirth: Accelerating deep neural network execution on mobile devices," in *Proc. Conf. Artif. Intell.*, 2018, pp. 1–9.

[187] G. Zhou et al., "Rocket launching: A universal and efficient framework for training well-performing light net," in *Proc. AAAI Conf. Artif. Intell.*, 2018, pp. 1–8.

[188] X. Jiao et al., "TinyBERT: Distilling BERT for natural language understanding," *Findings Assoc. Comput. Linguistics Findings ACL, EMNLP*, Sep. 2019, pp. 4163–4174.

[189] A. Chaulwar et al., "Extreme compression of sentence-transformer ranker models: Faster inference, longer battery life, and less storage on edge devices," 2022, *arXiv:2207.12852*.

[190] Y. Zhang, T. Xiang, T. M. Hospedales, and H. Lu, "Deep mutual learning," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 4320–4328.

[191] S. I. Mirzadeh, M. Farajtabar, A. Li, N. Levine, A. Matsukawa, and H. Ghasemzadeh, "Improved knowledge distillation via teacher assistant," in *Proc. AAAI Conf. Artif. Intell.*, 2020, vol. 34, no. 4, pp. 5191–5198.

[192] L. Yuan, F. E. Tay, G. Li, T. Wang, and J. Feng, "Revisiting knowledge distillation via label smoothing regularization," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2020, pp. 3903–3911.

[193] T. Wang, J.-Y. Zhu, A. Torralba, and A. A. Efros, "Dataset distillation," 2018, *arXiv:1811.10959*.

[194] O. Bohdal, Y. Yang, and T. Hospedales, "Flexible dataset distillation: Learn labels instead of images," 2020, *arXiv:2006.08572*.

[195] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, pp. 2736–2744.

[196] A. Gordon et al., "MorphNet: Fast & simple resource-constrained structure learning of deep networks," in *Proc. Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 1586–1595.

[197] J. Guo and M. Potkonjak, "Pruning filters and classes: Towards on-device customization of convolutional neural networks," in *Proc. 1st Int. Workshop Deep Learn. Mobile Syst. Appl.*, 2017, pp. 13–17.

[198] M. J. Shafiee, F. Li, B. Chwyl, and A. Wong, "SquishedNets: Squishing SqueezeNet further for edge device scenarios via deep evolutionary synthesis," 2017, *arXiv:1711.07459*.

[199] A. Wong, M. Famuori, M. J. Shafiee, F. Li, B. Chwyl, and J. Chung, "YOLO nano: A highly compact you only look once convolutional neural network for object detection," 2019, *arXiv:1910.01271*.

[200] S. Bhattacharya and N. D. Lane, "Sparsification and separation of deep learning layers for constrained resource inference on wearables," in *Proc. 14th ACM Conf. Embedded Netw. Sensor Syst.*, Nov. 2016, pp. 176–189.

[201] P. Maji, D. Bates, A. Chadwick, and R. Mullins, "ADaPT: Optimizing CNN inference on IoT and mobile devices using approximately separable 1-D kernels," in *Proc. 1st Int. Conf. Internet Things Mach. Learn.*, Oct. 2017, p. 43.

[202] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 1269–1277.

[203] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," 2014, *arXiv:1405.3866*.

[204] E. J. Crowley, G. Gray, and A. J. Storkey, "Moonshine: Distilling with cheap convolutions," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 2888–2898.

[205] S. Zagoruyko and N. Komodakis, "Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer," 2016, *arXiv:1612.03928*.

[206] B. B. Sau and V. N. Balasubramanian, "Deep model compression: Distilling knowledge from noisy teachers," 2016, *arXiv:1610.09650*.

[207] P.-K. Tsung, S.-F. Tsai, A. Pai, S.-J. Lai, and C. Lu, "High performance deep neural network on low cost mobile GPU," in *Proc. IEEE Int. Conf. Consum. Electron. (ICCE)*, Jan. 2016, pp. 69–70.

[208] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "MCDNN: An approximation-based execution framework for deep stream processing under resource constraints," in *Proc. 14th Annu. Int. Conf. Mobile Syst., Appl., Services*, Jun. 2016, pp. 123–136.

[209] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, "NoScope: Optimizing neural network queries over video at scale," *Proc. VLDB Endowment*, vol. 10, no. 11, pp. 1586–1597, 2017.

[210] E. Park et al., "Big/little deep neural network for ultra-low power inference," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth.*, 2015, pp. 124–132.

[211] H. Tann, S. Hashemi, R. I. Bahar, and S. Reda, "Runtime configurable deep neural networks for energy-accuracy trade-off," in *Proc. 11th IEEE/ACM/IFIP Int. Conf. Hardw./Softw. Codesign Syst. Synth.*, Oct. 2016, pp. 1–10.

[212] S. Teerapittayanon, B. McDanel, and H. T. Kung, "BranchyNet: Fast inference via early exiting from deep neural networks," in *Proc. 23rd Int. Conf. Pattern Recognit. (ICPR)*, Dec. 2016, pp. 2464–2469.

[213] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama, "Adaptive neural networks for efficient inference," 2017, *arXiv:1702.07811*.

[214] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 328–339.

[215] L. Li, K. Ota, and M. Dong, "Deep learning for smart industry: Efficient manufacture inspection system with fog computing," *IEEE Trans. Ind. Informat.*, vol. 14, no. 10, pp. 4665–4673, Oct. 2018.

[216] C. Lo, Y.-Y. Su, C.-Y. Lee, and S.-C. Chang, "A dynamic deep neural network design for efficient workload allocation in edge computing," in *Proc. IEEE Int. Conf. Comput. Design (ICCD)*, Nov. 2017, pp. 273–280.

[217] X. Wang, F. Yu, Z. Y. Dou, T. Darrell, and J. E. Gonzalez, "SkipNet: Learning dynamic routing in convolutional networks," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, Lecture Notes in Computer Science, vol. 11217, 2018, pp. 420–436.

[218] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "Envision: A 0.26-to-10 TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28 nm FDSOI," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2017, pp. 246–247.

[219] M. Parsa, P. Panda, S. Sen, and K. Roy, "Staged inference using conditional deep learning for energy efficient real-time smart diagnosis," in *Proc. 39th Annu. Int. Conf. IEEE Eng. Med. Biol. Soc. (EMBC)*, Jul. 2017, pp. 78–81.

[220] S. Cao et al., "SeerNet: Predicting convolutional neural network feature-map sparsity through low-bit quantization," in *Proc. Conf. Comput. Vis. Patt. Recognit.*, Jun. 2019, pp. 11216–11225.

[221] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking atrous convolution for semantic image segmentation," 2017, *arXiv:1706.05587*.

[222] A. Yazdanbakhsh, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, "GANAX: A unified MIMD-SIMD acceleration for generative adversarial networks," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 650–661.

[223] E. Elsen, M. Dukhan, T. Gale, and K. Simonyan, "Fast sparse ConvNets," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2020, pp. 14629–14638.

[224] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol.*, vol. 1, Jun. 2019, pp. 4171–4186.

[225] T. Gale, E. Elsen, and S. Hooker, "The state of sparsity in deep neural networks," 2019, *arXiv:1902.09574*.

[226] T. Wolf et al., "Transformers: State-of-the-art natural language processing," in *Proc. Conf. Empirical Methods Natural Lang. Process., Syst. Demonstrations. Assoc. Comput. Linguistics*, 2020, pp. 38–45.

[227] H. Wang, Z. Zhang, and S. Han, "SpAtten: Efficient sparse attention architecture with cascade token and head pruning," 2020, *arXiv:2012.09852*.

[228] S. Narang, E. Elsen, G. Diamos, and S. Sengupta, "Exploring sparsity in recurrent neural networks," 2017, *arXiv:1704.05119*.

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

[230] U. Gupta et al., "MASR: A modular accelerator for sparse RNNs," in *Proc. Int. Parallel Archit. Compilation Techn.*, 2019, pp. 1–14.

[231] Q. Yang, J. Mao, Z. Wang, and H. Li, "DASNet: Dynamic activation sparsity for neural network efficiency improvement," 2019, *arXiv:1909.06964*.

[232] J. Lee, J. Lee, D. Han, J. Lee, G. Park, and H.-J. Yoo, "LNPU: A 25.3 TFLOPS/W sparse deep-neural-network learning processor with fine-grained mixed precision of FP8–FP16," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2019, pp. 142–144.

[233] S. Zhang et al., "Cambricon-X: An accelerator for sparse neural networks," in *Proc. Int. Symp. Microarchitecture*, 2016, pp. 1–12.

[234] Z.-G. Liu, P. N. Whatmough, and M. Mattina, "Systolic tensor array: An efficient structured-sparse GEMM accelerator for mobile CNN inference," *IEEE Comput. Archit. Lett.*, vol. 19, no. 1, pp. 34–37, Jan. 2020.

[235] D. Kim, J. Ahn, and S. Yoo, "ZeNA: Zero-aware neural network accelerator," *IEEE Des. Test*, vol. 35, no. 1, pp. 39–46, Feb. 2018.

[236] H. T. Kung, B. McDanel, and S. Q. Zhang, "Adaptive tiling: Applying fixed-size systolic arrays to sparse convolutional neural networks," in *Proc. 24th Int. Conf. Pattern Recognit. (ICPR)*, Aug. 2018, pp. 1006–1011.

[237] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE J. Emerging Sel. Topics Circuits Syst.*, vol. 9, no. 2, pp. 292–308, Jun. 2019.

[238] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan, "PermDNN: Efficient compressed DNN architecture with permuted diagonal matrices," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 189–202.

[239] H. T. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2019, pp. 821–834.

[240] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh, "SnaPEA: Predictive early activation for reducing computation in deep convolutional neural networks," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 662–673.

[241] M. Song, J. Zhao, Y. Hu, J. Zhang, and T. Li, "Prediction based execution on deep neural networks," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 752–763.

[242] X. Zhou et al., "Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *Proc. Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2018, pp. 15–28.

[243] S. Han et al., "EIE: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 243–254, 2016.

[244] A. Parashar, "SCNN: An accelerator for compressed-sparse convolutional neural networks," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 27–40, 2017.

[245] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 1–13, 2016.

[246] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "SparTen: A sparse tensor accelerator for convolutional neural networks," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 151–165.

[247] R. Dorrance, F. Ren, and D. Marković, "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-BLAS on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2014, pp. 161–170.

[248] A. Aimar et al., "NullHop: A flexible convolutional neural network accelerator based on sparse representations of feature maps," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 3, pp. 644–656, Mar. 2019.

[249] J. Li et al., "SqueezeFlow: A sparse CNN accelerator exploiting concise convolution rules," *IEEE Trans. Comput.*, vol. 68, no. 11, pp. 1663–1677, Nov. 2019.

[250] Y.-H. Chen, T. Krishna, J.-S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Nov. 2016.

[251] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher, "UCNN: Exploiting computational reuse in deep neural networks via weight repetition," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 674–687.

[252] M. M. H. Shuvo, T. Titirsha, N. Amin, and S. K. Islam, "Energy harvesting in implantable and wearable medical devices for enduring precision healthcare," *Energies*, vol. 15, no. 20, p. 7495, Oct. 2022.

[253] *Jetson TX2 Module*. Accessed: Feb. 1, 2022. [Online]. Available: https://developer.nvidia.com/embedded/jetson-tx2

[254] *Intel® Edison Development Platform*. Accessed: Feb. 6, 2022. [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/edison/sb/edison_pb_331179002.pdf

[255] M. Abadi et al., "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Operating Syst. Design Implement.*, 2016, pp. 265–283.

[256] Y. Jia et al., "Caffe: Convolutional architecture for fast feature embedding," in *Proc. Int. Conf. Multimedia*, 2014, pp. 675–678.

[257] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 8026–8037.

[258] *Intel® Distribution of OpenVINO™ Toolkit*. Accessed: Feb. 8, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html

[259] M. Alzantot, Y. Wang, Z. Ren, and M. B. Srivastava, "RSTensorFlow: GPU enabled TensorFlow for deep learning on commodity Android devices," in *Proc. 1st Int. Workshop Deep Learn. Mobile Syst. Appl.*, 2017, pp. 7–12.

[260] *Open Neural Network Exchange*. Accessed: Feb. 16, 2022. [Online]. Available: https://onnx.ai/index.html

[261] X. Ma et al., "PCONV: The missing but desirable sparsity in DNN weight pruning for real-time execution on mobile devices," in *Proc. AAAI Conf. Artif. Intell.*, Apr. 2020, vol. 34, no. 4, pp. 5117–5124.

[262] W. Niu et al., "PatDNN: Achieving real-time DNN execution on mobile devices with pattern-based weight pruning," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2020, pp. 907–922.

[263] H. Guan et al., "CoCoPIE," *Commun. ACM*, vol. 64, no. 6, pp. 62–68, Jun. 2021.

[264] J. Lin, W.-M. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, "MCUNet: Tiny deep learning on IoT devices," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, Jul. 2020, pp. 11711–11722.

[265] *Deep Learning HDL Toolbox*. Accessed: Feb. 5, 2022. [Online]. Available: https://www.mathworks.com/products/deep-learning-hdl.html

[266] *X-CUBE-AI, AI Expansion Pack for STM32CubeMX*. Accessed: Feb. 7, 2022. [Online]. Available: https://www.st.com/en/embedded-software/x-cube-ai.html

[267] *DNNDK User Guide*. Accessed: Feb. 10, 2022. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug1327-dnndk-user-guide.pdf

[268] *NVIDIA TensorRT*. Accessed: Feb. 14, 2022. [Online]. Available: https://developer.nvidia.com/tensorrt

[269] *CEVA Deep Neural Network (CDNN)*. Accessed: Feb. 4, 2022. [Online]. Available: https://www.ceva-dsp.com/product/ceva-deep-neural-network-cdnn/

[270] *Qualcomm Neural Processing SDK for AI*. Accessed: Dec. 29, 2021. [Online]. Available: https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk

[271] *Stratus High-Level Synthesis*. Accessed: Jan. 13, 2022. [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html

[272] *Deploy Machine Learning Models on Mobile and IoT Devices*. Accessed: Feb. 10, 2022. [Online]. Available: https://www.tensorflow.org/lite

[273] *Embedded Learning Library (ELL)*. Accessed: Jan. 12, 2022. [Online]. Available: https://microsoft.github.io/ELL/

[274] S. Liu, B. Ren, X. Shen, and Y. Wang, "CoCoPIE: Making mobile AI sweet as PIE—Compression-compilation co-design goes a long way," 2020, *arXiv:2003.06700*.

[275] Y. Cai et al., "YOLObile: Real-time object detection on mobile devices via compression-compilation co-design," in *Proc. AAAI Conf. Artif. Intell.*, Sep. 2020, vol. 35, no. 2, pp. 955–963.

[276] S. S. L. Oskouei, H. Golestani, M. Hashemi, and S. Ghiasi, "CNNdroid: GPU-accelerated execution of trained deep convolutional neural networks on Android," in *Proc. 24th ACM Int. Conf. Multimedia*, Oct. 2016, pp. 1201–1205.

[277] S. Rizvi, G. Cabodi, D. Patti, and G. Francini, "GPGPU accelerated deep object classification on a heterogeneous mobile platform," *Electronics*, vol. 5, no. 4, p. 88, Dec. 2016.

[278] Q. Cao, N. Balasubramanian, and A. Balasubramanian, "MobiRNN: Efficient recurrent neural network execution on mobile GPU," in *Proc. 1st Int. Workshop Deep Learn. Mobile Syst. Appl.*, 2017, pp. 1–6.

[279] H. Guihot, "RenderScript," in *Pro Android Apps Performance Optimization*. New York, NY, USA: Apress, 2012, pp. 231–263.

[280] M. Motamedi, D. Fong, and S. Ghiasi, "Cappuccino: Efficient CNN inference software synthesis for mobile system-on-chips," *IEEE Embedded Syst. Lett.*, vol. 11, no. 1, pp. 9–12, Mar. 2019.

[281] L. N. Huynh, R. K. Balan, and Y. Lee, "DeepSense: A GPU-based deep convolutional neural network framework on commodity mobile devices," in *Proc. Workshop Wearable Syst. Appl.*, 2016, pp. 25–30.

[282] S. Rallapalli et al., "Are very deep neural networks feasible on mobile devices?" Univ. Southern California, Los Angeles, CA, USA, Tech. Rep., 2016, pp. 916–965.

[283] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 779–788.

[284] N. D. Lane et al., "DeepX: A software accelerator for low-power deep learning inference on mobile devices," in *Proc. Int. Conf. Inf. Process. Sensor Netw.*, 2016, p. 23.

[285] N. Lane, S. Bhattacharya, A. Mathur, C. Forlivesi, and F. Kawsar, "DXTK: Enabling resource-efficient deep learning on mobile and embedded devices with the DeepX toolkit," in *Proc. 8th EAI Int. Conf. Mobile Comput., Appl. Services*, 2016, pp. 98–107.

[286] Z. Du et al., "ShiDianNao: Shifting vision processing closer to the sensor," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 3, pp. 92–104, 2015.

[287] Y. Chen, T. Chen, Z. Xu, N. Sun, and O. Temam, "Diannao family: Energy-efficient hardware accelerators for machine learning," *Commun. ACM*, vol. 59, no. 11, pp. 105–112, 2016.

[288] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "NeuFlow: A runtime reconfigurable dataflow processor for vision," in *Proc. CVPR Workshops*, Jun. 2011, pp. 109–116.

[289] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects," *ACM Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, vol. 53, pp. 461–475, Mar. 2018.

[290] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "DeepSpeed," in *Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2020, pp. 3505–3506.

[291] J. Ren et al., "ZeRO-Offload: Democratizing billion-scale model training," in *Proc. USENIX Annu. Tech. Conf.*, Jul. 2021, pp. 551–564.

[292] H. Tang et al., "1-bit Adam: Communication efficient large-scale training with Adam's convergence speed," in *Proc. Int. Conf. Mach. Learn.*, Feb. 2021, pp. 10118–10129.

[293] Y. Lu, C. Li, M. Zhang, C. De Sa, and Y. He, "Maximizing communication efficiency for large-scale training via 0/1 Adam," 2022, *arXiv:2202.06009*.

[294] C. Li, A. A. Awan, H. Tang, S. Rajbhandari, and Y. He, "1-bit Lamb: Communication efficient large-scale large-batch training with Lamb's convergence speed," 2021, *arXiv:2104.06069*.

[295] Z. Yao, R. Y. Aminabadi, M. Zhang, X. Wu, C. Li, and Y. He, "ZeroQuant: Efficient and affordable post-training quantization for large-scale transformers," 2022, *arXiv:2206.01861*.

[296] Y. Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. Adv. Neural Inf. Process. Syst.*, Nov. 2018, pp. 1–10.

[297] A. Sergeev and M. D. Balso, "Horovod: Fast and easy distributed deep learning in TensorFlow," 2018, *arXiv:1802.05799*.

[298] J. Pan, W. Liu, and J. Zhou, "Benchmark tests of convolutional neural network and graph convolutional network on HorovodRunner enabled spark clusters," 2020, *arXiv:2005.05510*.

[299] M. Baines et al., "FairScale: A general purpose modular PyTorch library for high performance and large scale training," 2021. [Online]. Available: https://github.com/facebookresearch/fairscale

[300] N. Shazeer et al., "Mesh-TensorFlow: Deep learning for supercomputers," in *Proc. Adv. Neural Inf. Process. Syst.*, Nov. 2018, pp. 1–10.

[301] P. Buchlovsky et al., "TF-replicator: Distributed machine learning for researchers," 2019, *arXiv:1902.00465*.

[302] J. J. Dai et al., "BigDL," in *Proc. ACM Symp. Cloud Comput.*, Nov. 2019, pp. 50–60.

[303] P. Moritz et al., "Ray: A distributed framework for emerging AI applications," in *Proc. 13th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, Oct. 2018, pp. 561–577.

[304] A. Alsalemi, A. Al-Kababji, Y. Himeur, F. Bensaali, and A. Amira, "Cloud energy micro-moment data

classification: A platform study," in *Proc. IEEE/ACM 13th Int. Conf. Utility Cloud Comput. (UCC)*, Dec. 2020, pp. 420–425.
[305] *Tensilica DNA Processor Family for on-Device AI*. Accessed: Jan. 20, 2022. [Online]. Available: https://ip.cadence.com/ai&CMP=TIP_AI1_IndTre_Arti_0918_PP
[306] *DesignWare ARC EV Processors for Embedded Vision*. Accessed: Feb. 21, 2022. [Online]. Available: https://www.synopsys.com/designware-ip/processor-solutions/ev-processors.html
[307] *AI for the Edge*. Accessed: Jan. 30, 2022. [Online]. Available: https://www.gyrfalcontech.ai/solutions/
[308] D. Han, J. Lee, J. Lee, and H.-J. Yoo, "A 1.32 TOPS/W energy efficient deep neural network learning processor with direct feedback alignment based heterogeneous core architecture," in *Proc. Symp. VLSI Circuits*, Jun. 2019, pp. C304–C305.
[309] J.-F. Zhang, C.-E. Lee, C. Liu, Y. S. Shao, S. W. Keckler, and Z. Zhang, "SNAP: A 1.67—21.55 TOPS/W sparse neural acceleration processor for unstructured sparse deep neural network inference in 16 nm CMOS," in *Proc. Symp. VLSI Circuits*, Jun. 2019, pp. C306–C307.
[310] Z. Yuan et al., "Sticker: A 0.41–62.1 TOPS/W 8 bit neural network processor with multi-sparsity compatible convolution arrays and online tuning acceleration for fully connected layers," in *Proc. IEEE Symp. VLSI Circuits*, Jun. 2018, pp. 33–34.
[311] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 553–564.
[312] *Products, Helping You Bring Local AI to Applications From Prototype to Production*. Accessed: Feb. 21, 2022. [Online]. Available: https://coral.ai/products/
[313] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck, "DeltaRNN: A power-efficient recurrent neural network accelerator," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2018, pp. 21–30.
[314] S. Wang et al., "C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs," in *Proc. Int. Symp. Field-Prog. Gate Arrays*, 2018, pp. 11–20.
[315] K. Guo et al., "Angel-eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 35–47, Jan. 2018.
[316] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: Mapping regular and irregular convolutional neural networks on FPGAs," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 2, pp. 326–342, Feb. 2019.
[317] M. Vestias, R. Policarpo Duarte, J. T. de Sousa, and H. Neto, "Lite-CNN: A high-performance architecture to execute CNNs in low density FPGAs," in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2018, pp. 393–399.
[318] B. Fleischer et al., "A scalable multi-TeraOPS deep learning processor core for AI trainina and inference," in *Proc. IEEE Symp. VLSI Circuits*, Jun. 2018, pp. 35–36.
[319] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *Proc. Int. Symp. Comput. Archit.*, 2017, pp. 1–12.
[320] S. Yin et al., "A high energy efficient reconfigurable hybrid neural network processor for deep learning applications," *IEEE J. Solid-State Circuits*, vol. 53, pp. 968–982, 2018.
[321] S. Yin et al., "A 1.06-to-5.09 TOPS/W reconfigurable hybrid-neural-network processor for deep learning applications," in *Proc. Symp. VLSI Circuits*, 2017, pp. C26–C27.
[322] D. Shin, J. Lee, J. Lee, J. Lee, and H.-J. Yoo, "DNPU: An energy-efficient deep-learning processor with heterogeneous multi-core architecture," *IEEE Micro*, vol. 38, no. 5, pp. 85–93, Sep. 2018.
[323] X. Fan, D. Wu, W. Cao, W. Luk, and L. Wang, "Stream processing dual-track CGRA for object inference," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 6, pp. 1098–1111, Jun. 2018.
[324] J. Pei et al., "Towards artificial general intelligence with hybrid tianjic chip architecture," *Nature*, vol. 572, no. 7767, pp. 106–111, 2019.
[325] B. Zimmer et al., "A 0.11 pJ/Op, 0.32–128 TOPS, scalable multi-chip-module-based deep neural network accelerator with ground-reference signaling in 16 nm," in *Proc. Symp. VLSI Circuits*, 2019, pp. C300–C301.
[326] Y. Chen et al., "DaDianNao: A machine-learning supercomputer," in *Proc. Int. Symp. Microarch.*, 2014, pp. 609–622.
[327] T. Fujii et al., "New generation dynamically reconfigurable processor technology for accelerating embedded AI applications," in *Proc. Symp. VLSI Circuits*, 2018, pp. 41–42.
[328] M. Liang, M. Chen, Z. Wang, and J. Sun, "A CGRA based neural network inference engine for deep reinforcement learning," in *Proc. IEEE Asia Pacific Conf. Circuits Syst. (APCCAS)*, Oct. 2018, pp. 540–543.
[329] J. Fowers et al., "A configurable cloud-scale DNN processor for real-time AI," in *Proc. Ann. Int. Symp. Comput. Arch.*, 2018, pp. 1–14.
[330] O. Shacham and M. Reynders. *Pixel Visual Core: Image Processing and Machine Learning on Pixel 2*. Accessed: Feb. 21, 2022. [Online]. Available: https://blog.google/products/pixel/pixel-visual-core-image-processing-and-machine-learning-pixel-2/
[331] *EyeQ® The System-on-Chip for Automotive Applications*. Accessed: Jan. 12, 2022. [Online]. Available: https://www.mobileye.com/our-technology/evolution-eyeq-chip/
[332] D. Rossi, I. Loi, F. Conti, G. Tagliavini, A. Pullini, and A. Marongiu, "Energy efficient parallel computing on the PULP platform with support for OpenMP," in *Proc. IEEE 28th Conv. Electr. Electron. Eng. (IEEEI)*, Dec. 2014, pp. 1–5.
[333] A. Garofalo, G. Tagliavini, F. Conti, D. Rossi, and L. Benini, "XpulpNN: Accelerating quantized neural networks on RISC-V processors through ISA extensions," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2020, pp. 186–191.
[334] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," 2017, *arXiv:1711.07128*.
[335] P.-E. Novac, G. B. Hacene, A. Pegatoquet, B. Miramond, and V. Gripon, "Quantization and deployment of deep neural networks on microcontrollers," *Sensors*, vol. 21, no. 9, p. 2984, Apr. 2021.
[336] P. Chi et al., "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 27–39, 2016.
[337] S. Ambrogio et al., "Equivalent-accuracy accelerated neural-network training using analogue memory," *Nature*, vol. 558, no. 7708, pp. 60–67, 2018.
[338] C. Eckert et al., "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *Proc. Int. Symp. Comput. Archit.*, 2018, pp. 383–396.
[339] S. Li et al., "SCOPE: A stochastic computing engine for DRAM-based in-situ accelerator," in *Proc. Int. Symp. Microarchitecture*, 2018, pp. 696–709.
[340] X. Guo et al., "Fast, energy-efficient, robust, and reproducible mixed-signal neuromorphic classifier based on embedded NOR flash memory technology," in *IEDM Tech. Dig.*, Dec. 2017, pp. 5–6.
[341] T. Tang, L. Xia, B. Li, Y. Wang, and H. Yang, "Binary convolutional neural network on RRAM," in *Proc. 22nd Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2017, pp. 782–787.
[342] P. Wang, Y. Ji, C. Hong, Y. Lyu, D. Wang, and Y. Xie, "SNrram: An efficient sparse neural network computation architecture based on resistive random-access memory," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, Jun. 2018, p. 106.
[343] L. Liang et al., "Crossbar-aware neural network pruning," *IEEE Access*, vol. 6, pp. 58324–58337, 2018.
[344] J. Lin, Z. Zhu, Y. Wang, and Y. Xie, "Learning the sparsity for ReRAM: Mapping and pruning sparse neural network for ReRAM based accelerator," in *Proc. 24th Asia South Pacific Design Autom. Conf.*, Jan. 2019, pp. 639–644.
[345] T.-H. Yang et al., "Sparse ReRAM engine: Joint exploration of activation and weight sparsity in compressed neural networks," in *Proc. Int. Symp. Comput. Archit.*, 2019, pp. 236–249.
[346] N. D. Lane, S. Bhattacharya, A. Mathur, P. Georgiev, C. Forlivesi, and F. Kawsar, "Squeezing deep learning into mobile and embedded devices," *IEEE Pervasive Comput.*, vol. 16, no. 3, pp. 82–88, Jul. 2017.
[347] V. Radu et al., "Multimodal deep learning for activity and context recognition," *Proc. ACM Interact., Mobile, Wearable Ubiquitous Technol.*, vol. 1, no. 4, pp. 1–27, Jan. 2018.
[348] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, and F. Kawsar, "An early resource characterization of deep learning on wearables, smartphones and Internet-of-Things devices," in *Proc. Int. Workshop Internet Things Appl.*, Nov. 2015, pp. 7–12.
[349] Y. Chen, S. Biookaghazadeh, and M. Zhao, "Exploring the capabilities of mobile devices in supporting deep learning," in *Proc. 4th ACM/IEEE Symp. Edge Comput.*, Nov. 2019, pp. 127–138.
[350] D. Li, T. Salonidis, N. V. Desai, and M. C. Chuah, "DeepCham: Collaborative edge-mediated adaptive deep learning for mobile object recognition," in *Proc. IEEE/ACM Symp. Edge Comput. (SEC)*, Oct. 2016, pp. 64–76.
[351] M. M. H. Shuvo, "Edge AI: Leveraging the full potential of deep learning," in *Recent Innovations in Artificial Intelligence and Smart Applications*. Cham, Switzerland: Springer, 2022, ch. 2, pp. 27–46.
[352] Y. Huang et al., "Task scheduling with optimized transmission time in collaborative cloud-edge learning," in *Proc. 27th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Jul. 2018, pp. 1–9.
[353] L. Valerio, A. Passarella, and M. Conti, "A communication efficient distributed learning framework for smart environments," *Pervasive Mobile Comput.*, vol. 41, pp. 46–68, Oct. 2017.
[354] O. Valery, P. Liu, and J.-J. Wu, "CPU/GPU collaboration techniques for transfer learning on mobile devices," in *Proc. IEEE 23rd Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2017, pp. 477–484.
[355] O. Valery, P. Liu, and J.-J. Wu, "Low precision deep learning training on mobile heterogeneous platform," in *Proc. 26th Euromicro Int. Conf. Parallel, Distrib. Netw.-Based Process. (PDP)*, Mar. 2018, pp. 109–117.
[356] M. Ware, E. Frank, G. Holmes, M. Hall, and I. H. Witten, "Interactive machine learning: Letting users build classifiers," *Int. J. Hum.-Comput. Stud.*, vol. 55, no. 3, pp. 281–292, Sep. 2001.
[357] S. Amershi, M. Cakmak, W. B. Knox, and T. Kulesza, "Power to the people: The role of humans in interactive machine learning," *AI Mag.*, vol. 35, no. 4, pp. 105–120, 2014.
[358] F. Shahmohammadi, A. Hosseini, C. E. King, and M. Sarrafzadeh, "Smartwatch based activity recognition using active learning," in *Proc. IEEE/ACM Int. Conf. Connected Health, Appl., Syst. Eng. Technol. (CHASE)*, Jul. 2017, pp. 321–329.
[359] B. McMahan, E. Moore, D. Ramage, S. Hampson, B. A. Y. Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial Intelligence and Statistics*. PMLR, 2017, pp. 1273–1282.
[360] Q. Yang, Y. Liu, Y. Cheng, Y. Kang, T. Chen, and H. Yu, "Federated learning," *Synth. Lectures Artif. Intell. Mach. Learn.*, vol. 13, no. 3, pp. 1–207, Dec. 2019.
[361] K. Bonawitz, P. Kairouz, B. McMahan, and

D. Ramage, "Federated learning and privacy," *Queue*, vol. 19, no. 5, pp. 87–114, Oct. 2021.
[362] R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 1310–1321.
[363] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," 2016, *arXiv:1610.05492*.
[364] H. Kim, J. Park, M. Bennis, and S.-L. Kim, "Blockchained on-device federated learning," *IEEE Commun. Lett.*, vol. 24, no. 6, pp. 1279–1283, Jun. 2020.
[365] Y. Mao, S. Yi, Q. Li, J. Feng, F. Xu, and S. Zhong, "A privacy-preserving deep learning approach for face recognition with edge computing," in *Proc. USENIX Workshop Hot Topics Edge Comput. (HotEdge)*, Jul. 2018, pp. 1–6.
[366] J. Wang, J. Zhang, W. Bao, X. Zhu, B. Cao, and P. S. Yu, "Not just privacy," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2018, pp. 2407–2416.
[367] S. A. Osia et al., "A hybrid deep learning architecture for privacy-preserving mobile analytics," *IEEE Internet Things J.*, vol. 7, no. 5, pp. 4505–4518, May 2020.
[368] A. Harlap et al., "PipeDream: Fast and efficient pipeline parallel DNN training," 2018, *arXiv:1806.03377*.
[369] H. Saghir, S. Choudhary, S. Eghbali, and C. Chung, "Factorization-aware training of transformers for natural language understanding on the edge," in *Proc. Interspeech*, Aug. 2021, pp. 4733–4737.
[370] S. S. Ogden and T. Guo, "MODI: Mobile deep inference made efficient by edge computing," in *Proc. USENIX Workshop Hot Topics Edge Comput.*, Jul. 2018, pp. 1–7.
[371] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "MoDNN: Local distributed mobile computing system for deep neural network," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 1396–1401.
[372] R. Stahl, Z. Zhao, D. Mueller-Gritschneder, A. Gerstlauer, and U. Schlichtmann, "Fully distributed deep learning inference on resource-constrained edge devices," in *Proc. Int. Conf. Embedded Comput. Syst.*, 2019, pp. 77–90.
[373] Z. Huai, B. Ding, H. Wang, M. Geng, and L. Zhang, "Towards deep learning on resource-constrained robots: A crowdsourcing approach with model partition," in *Proc. IEEE SmartWorld, Ubiquitous Intell., Comput., Adv., Trusted Comput., Scalable Comput., Commun., Cloud, Big Data Comput., Internet People Smart City Innov. (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, Aug. 2019, pp. 989–994.
[374] H. Li, C. Hu, J. Jiang, Z. Wang, Y. Wen, and W. Zhu, "JALAD: Joint accuracy- and latency-aware deep structure decoupling for edge-cloud execution," in *Proc. IEEE 24th Int. Conf. Parallel Distrib. Syst.*, Dec. 2018, pp. 671–678.
[375] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2348–2359, Nov. 2018.
[376] L. Zeng, X. Chen, Z. Zhou, L. Yang, and J. Zhang, "CoEdge: Cooperative DNN inference with adaptive workload partitioning over heterogeneous edge devices," *IEEE/ACM Trans. Netw.*, vol. 29, no. 2, pp. 595–608, Apr. 2021.
[377] S. Naveen, M. R. Kounte, and M. R. Ahmed, "Low latency deep learning inference model for distributed intelligent IoT edge clusters," *IEEE Access*, vol. 9, pp. 160607–160621, 2021.
[378] E. Li, Z. Zhou, and X. Chen, "Edge intelligence," in *Proc. Workshop Mobile Edge Commun.*, Aug. 2018, pp. 31–36.
[379] X. Hou, Y. Guan, T. Han, and N. Zhang, "DistrEdge: Speeding up convolutional neural network inference on distributed edge devices," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2022, pp. 1097–1107.
[380] C. Hu and B. Li, "Distributed inference with deep learning models across heterogeneous edge devices," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, May 2022, pp. 330–339.
[381] T. Ge and F. Wei, "EdgeFormer: A parameter-efficient transformer for on-device Seq2Seq generation," 2022, *arXiv:2202.07959*.
[382] T. Ge, H. Xia, X. Sun, S.-Q. Chen, and F. Wei, "Lossless acceleration for Seq2seq generation with aggressive decoding," 2022, *arXiv:2205.10350*.
[383] D. Wen, H. Han, and A. K. Jain, "Face spoof detection with image distortion analysis," *IEEE Trans. Inf. Forensics Security*, vol. 10, no. 4, pp. 746–761, Apr. 2015.
[384] B. Fan, X. Liu, X. Su, P. Hui, and J. Niu, "EmgAuth: An EMG-based smartphone unlocking system using Siamese network," in *Proc. IEEE Int. Conf. Pervasive Comput. Commun. (PerCom)*, Mar. 2020, pp. 1–10.
[385] S. Chen, Y. Liu, X. Gao, and Z. Han, "MobileFaceNets: Efficient CNNs for accurate real-time face verification on mobile devices," in *Biometric Recognition*. Cham, Switzerland: Springer, 2018, pp. 428–438.
[386] C. N. Duong, K. G. Quach, I. Jalata, N. Le, and K. Luu, "MobiFace: A lightweight deep learning face recognition on mobile devices," in *Proc. IEEE 10th Int. Conf. Biometrics Theory, Appl. Syst. (BTAS)*, Sep. 2018, pp. 1–6.
[387] M. M. H. Shuvo, N. Ahmed, K. Nouduri, and K. Palaniappan, "A hybrid approach for human activity recognition with support vector machine and 1D convolutional neural network," in *Proc. IEEE Appl. Imag. Pattern Recognit. Workshop (AIPR)*, Oct. 2020, pp. 1–5.
[388] S. Bhattacharya and N. D. Lane, "From smart to deep: Robust activity recognition on smartwatches using deep learning," in *Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops (PerCom Workshops)*, Mar. 2016, pp. 1–6.
[389] B. Almaslukh, J. A. Muhtadi, and A. M. Artoli, "A robust convolutional neural network for online smartphone-based human activity recognition," *J. Intell. Fuzzy Syst.*, vol. 35, no. 2, pp. 1609–1620, Aug. 2018.
[390] B. Almaslukh, A. Artoli, and J. Al-Muhtadi, "A robust deep learning approach for position-independent smartphone-based human activity recognition," *Sensors*, vol. 18, no. 11, p. 3726, Nov. 2018.
[391] P. Sundaramoorthy, G. K. Gudur, M. R. Moorthy, R. N. Bhandari, and V. Vijayaraghavan, "HARNet: Towards on-device incremental learning using deep ensembles on constrained devices," in *Proc. 2nd Int. Workshop Embedded Mobile Deep Learn.*, Jun. 2018, pp. 31–36.
[392] S. Yao, S. Hu, Y. Zhao, A. Zhang, and T. Abdelzaher, "DeepSense: A unified deep learning framework for time-series mobile sensing data processing," in *Proc. 26th Int. Conf. World Wide Web*, Apr. 2017, pp. 351–360.
[393] A. Mathur, N. D. Lane, S. Bhattacharya, A. Boran, C. Forlivesi, and F. Kawsar, "DeepEye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware," in *Proc. 15th Annu. Int. Conf. Mobile Syst., Appl., Services*, Jun. 2017, pp. 68–81.
[394] P. Wang, Q. Hu, Z. Fang, C. Zhao, and J. Cheng, "DeepSearch: A fast image search framework for mobile devices," *ACM Trans. Multimedia Comput., Commun., Appl.*, vol. 14, no. 1, pp. 1–22, Jan. 2018.
[395] X. Zeng, K. Cao, and M. Zhang, "MobileDeepPill: A small-footprint mobile deep learning system for recognizing unconstrained pill images," in *Proc. 15th Annu. Int. Conf. Mobile Syst., Appl., Services*, Jun. 2017, pp. 56–67.
[396] C. Streiffer, R. Raghavendra, T. Benson, and M. Srivatsa, "DarNet: A deep learning solution for distracted driving detection," in *Proc. USENIX Middleware Conf. Ind. Track*, Apr. 2017, pp. 22–28.
[397] N. D. Lane, P. Georgiev, and L. Qendro, "DeepEar: Robust smartphone audio sensing in unconstrained acoustic environments using deep learning," in *Proc. ACM Int. Joint Conf. Pervasive Ubiquitous Comput. (UbiComp)*, 2015, pp. 283–294.
[398] P. Georgiev, S. Bhattacharya, N. D. Lane, and C. Mascolo, "Low-resource multi-task audio sensing for mobile and embedded devices via shared deep neural network representations," in *Proc. ACM Interact., Mobile, Wearable Ubiquitous Technol.*, 2017, vol. 1, no. 3, p. 50.
[399] B. Kim, Y. Jeon, H. Park, D. Han, and Y. Baek, "Design and implementation of the vehicular camera system using deep neural network compression," in *Proc. 1st Int. Workshop Deep Learn. Mobile Syst. Appl. (EMDL)*, Apr. 2017, pp. 25–30.
[400] X. Xu, S. Yin, and P. Ouyang, "Fast and low-power behavior analysis on vehicles using smartphones," in *Proc. 6th Int. Symp. Next Gener. Electron. (ISNE)*, May 2017, pp. 1–4.
[401] B. Taylor, V. S. Marco, W. Wolff, Y. Elkhatib, and Z. Wang, "Adaptive deep learning model selection on embedded systems," *ACM SIGPLAN Notices*, vol. 53, no. 6, pp. 31–43, Dec. 2018.
[402] L. Lai and N. Suda, "Enabling deep learning at the IoT edge," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2018, p. 135.
[403] T. Zhu, L. Kuang, K. Li, J. Zeng, P. Herrero, and P. Georgiou, "Blood glucose prediction in type 1 diabetes using deep learning on the edge," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2021, pp. 1–5.
[404] M. P. Véstias, R. P. Duarte, J. T. de Sousa, and H. C. Neto, "Fast convolutional neural networks in low density FPGAs using zero-skipping and weight pruning," *Electronics*, vol. 8, no. 11, p. 1321, Nov. 2019.
[405] J.-H. Luo, H. Zhang, H.-Y. Zhou, C.-W. Xie, J. Wu, and W. Lin, "ThiNet: Pruning CNN filters for a thinner net," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 41, no. 10, pp. 2525–2538, Oct. 2019.
[406] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "FP-BNN: Binarized neural network on FPGA," *Neurocomputing*, vol. 275, pp. 1072–1086, Jan. 2018.
[407] J. Faraone et al., "AddNet: Deep neural networks using FPGA-optimized multipliers," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 1, pp. 115–128, Jan. 2020.
[408] F. Shang, J. Lai, J. Chen, W. Xia, and H. Liu, "A model compression based framework for electrical equipment intelligent inspection on edge computing environment," in *Proc. Int. Cloud Comput. Big Data Anal.*, Apr. 2021, pp. 406–410.
[409] B. Wang, F. Ma, L. Ge, H. Ma, H. Wang, and M. A. Mohamed, "Icing-EdgeNet: A pruning lightweight edge intelligent method of discriminative driving channel for ice thickness of transmission lines," *IEEE Trans. Instrum. Meas.*, vol. 70, pp. 1–12, 2021.
[410] Q. Wen, "Time series data augmentation for deep learning: A survey," 2020 *arXiv:2002.12478*.
[411] P. Chlap, H. Min, N. Vandenberg, J. Dowling, L. Holloway, and A. Haworth, "A review of medical image data augmentation techniques for deep learning applications," *J. Med. Imag. Radiat. Oncol.*, vol. 65, no. 5, pp. 545–563, Aug. 2021.
[412] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1798–1828, Aug. 2013.
[413] M. Blott, N. J. Fraser, and G. Gambardella, "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Trans. Reconfigurable Tech. Syst.*, vol. 11, no. 3, pp. 1–23, 2018.
[414] H. Sharma, J. Park, D. Mahajan, and E. Amaro, "From high-level deep neural models to FPGAs," in *Proc. Annu. Int. Symp. Microarchitecture*, 2016, p. 17.
[415] X. Zhang, J. Wang, C. Zhu, Y. Lin, and J. Xiong, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," in *Proc. Int. Conf. Comput.-Aided Design*, May 2018, p. 56.
[416] N. Srivastava, H. Rong, P. Barua, and G. Feng,

"T2S-tensor: Productively generating high-performance spatial hardware for dense tensor computations," in *Proc. Int. Symp. Field-Prog. Custom Comput. Mach.*, May 2019, pp. 181–189.

[417] Y.-H. Lai, "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proc. Int. Symp. Field-Program. Gate Arrays*, 2019, pp. 242–251.

[418] H. Cai, L. Zhu, and S. Han, "ProxylessNAS: Direct neural architecture search on target task and hardware," 2018, *arXiv:1812.00332*.

[419] S. Venkataraman and A. Ranjan, "ScaleDeep: A scalable compute architecture for learning and evaluating deep networks," in *Proc. Int. Symp. Comput. Archit.*, Feb. 2017, pp. 13–26.

[420] L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "HyPar: Towards hybrid parallelism for deep learning accelerator array," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2019, pp. 56–68.

[421] P. Ren, "A survey of deep active learning," 2020, *arXiv:2009.00236*.

[422] Z. Chen and B. Liu, "Lifelong machine learning," *Synth. Lectures Artif. Intell. Mach. Learn.*, vol. 12, no. 3, pp. 1–207, 2018.

[423] Y. Deng, X. Chen, G. Zhu, Y. Fang, Z. Chen, and X. Deng, "Actions at the edge: Jointly optimizing the resources in multi-access edge computing," *IEEE Wireless Commun.*, vol. 29, no. 2, pp. 192–198, Apr. 2022.

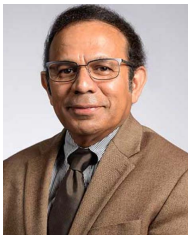
[424] A. Adadi and M. Berrada, "Peeking inside the black-box: A survey on explainable artificial intelligence (XAI)," *IEEE Access*, vol. 6, pp. 52138–52160, 2018.

ABOUT THE AUTHORS

Md. Maruf Hossain Shuvo (Graduate Student Member, IEEE) received the B.Sc. degree in electronics and communication engineering (ECE) from the Khulna University of Engineering & Technology (KUET), Khulna, Bangladesh, in 2014, and the M.S. degree in electrical engineering from the University of Missouri, Columbia, MO, USA, in 2021, where he is currently working toward the Ph.D. degree in electrical and computer engineering.

He was a Lecturer and an Assistant Professor with the ECE Department, KUET, from 2015 to 2019. His research interests include machine learning, biomedical signal and image analysis, integrated circuit design, deep learning, edge intelligence, and biomedical electronics.

Mr. Shuvo was a recipient of the University Gold Medal from KUET and the Prime Minister Gold Medal from the University Grants Commission of Bangladesh.



Syed Kamrul Islam (Senior Member, IEEE) received the B.Sc. degree in electrical and electronic engineering from the Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh, in 1983, and the M.S. and Ph.D. degrees in electrical and systems engineering from the University of Connecticut, Storrs, CT, USA, in 1987 and 1994, respectively.

He is currently a Professor and the Chair of the Department of Electrical Engineering and Computer Science, University of Missouri, Columbia, MO, USA. Prior to joining the Department of Electrical Engineering and Computer Science, University of Missouri, in July 2018, he was the James W. McConnell Professor and the Associate Head of the Department of Electrical Engineering and Computer Science, The University of Tennessee, Knoxville, TN, USA. His research interests include analog/mixed-signal integrated circuits, semiconductor devices, nanotechnology, biomicroelectronics, and monolithic sensors.

Dr. Islam, in recognition of his teaching, research, and related efforts at the University of Tennessee, received the John W. Fisher Professorship, the Eta Kappa Nu Outstanding Teacher Award, the Moses E. and Mayme Brooks Distinguished Professor Award, the College of Engineering Research Fellow Award, The Gonzalez Family Award for Excellence in Teaching, the Tickle College of Engineering Teaching Fellow, the University of Tennessee Citation for Research and Creative Achievement, the Electrical and Computer Engineering Faculty of the Year Award, and the Alexander Prize.

Jianlin Cheng (Member, IEEE) received the Ph.D. degree in information and computer science from the University of California at Irvine, Irvine, CA, USA, in 2006.

He is currently the William and Nancy Thompson Professor with the Department of Electrical Engineering and Computer Science, University of Missouri, Columbia, MO, USA. He is one of the top 100 AI leaders in drug discovery and healthcare in the world according to Deep Knowledge Analytics, 2019. His research interests include bioinformatics, machine learning, and artificial intelligence.

Dr. Cheng was a recipient of the NSF CAREER Award and the MU College of Engineering's Junior and Senior Faculty Research Awards.

Bashir I. Morshed (Senior Member, IEEE) received the B.Sc. degree in electrical and electronics engineering from the Bangladesh University of Engineering and Technology, Dhaka, Bangladesh, in 2001, the M.Sc. degree in electrical and computer engineering from the University of Windsor, Windsor, ON, Canada, in 2004, and the Ph.D. degree in electrical and computer engineering from Carleton University, Ottawa, ON, Canada, in 2010.

He has been with the Electrical and Computer Engineering Department, The University of Memphis, Memphis, TN, USA, since 2011. He has been an Associate Professor with the Computer Science Department, Texas Tech University, Lubbock, TX, USA, since 2020.

Dr. Morshed was a recipient of the prestigious Canadian Commonwealth Scholarship in 2002.