information

Review

Advancements in On-Device Deep Neural Networks

Kavya Saravanan [1,2] and Abbas Z. Kouzani [1,*]

[1] School of Engineering, Deakin University, Geelong, VIC 3216, Australia
[2] Department of Sensor and Biomedical Technology, Vellore Institute of Technology, Vellore 632014, India
* Correspondence: kouzani@deakin.edu.au

Abstract: In recent years, rapid advancements in both hardware and software technologies have resulted in the ability to execute artificial intelligence (AI) algorithms on low-resource devices. The combination of high-speed, low-power electronic hardware and efficient AI algorithms is driving the emergence of on-device AI. Deep neural networks (DNNs) are highly effective AI algorithms used for identifying patterns in complex data. DNNs, however, contain many parameters and operations that make them computationally intensive to execute. Accordingly, DNNs are usually executed on high-resource backend processors. This causes an increase in data processing latency and energy expenditure. Therefore, modern strategies are being developed to facilitate the implementation of DNNs on devices with limited resources. This paper presents a detailed review of the current methods and structures that have been developed to deploy DNNs on devices with limited resources. Firstly, an overview of DNNs is presented. Next, the methods used to implement DNNs on resource-constrained devices are explained. Following this, the existing works reported in the literature on the execution of DNNs on low-resource devices are reviewed. The reviewed works are classified into three categories: software, hardware, and hardware/software co-design. Then, a discussion on the reviewed approaches is given, followed by a list of challenges and future prospects of on-device AI, together with its emerging applications.

Keywords: artificial intelligence; deep neural networks; resource-constrained devices; on-device AI

Citation: Saravanan, K.; Kouzani, A.Z. Advancements in On-Device Deep Neural Networks. *Information* **2023**, *14*, 470. https://doi.org/10.3390/info14080470

Academic Editors: Lorenzo Carnevale and Massimo Villari

Received: 22 June 2023
Revised: 20 July 2023
Accepted: 17 August 2023
Published: 21 August 2023

1. Introduction

Executing artificial intelligence algorithms on hardware devices with limited resources, such as low-end microcontrollers, is becoming a reality because of the increasing power and decreasing cost of electronic hardware technology, as well as the creation of more efficient AI algorithms. In emerging research fields such as on-device AI, data can be analyzed where sensors generate data instead of backend servers. This has created new opportunities for advanced AI applications such as autonomous vehicles. Moreover, the rise of modern methods for on-device inference and the creation of pre-trained AI algorithms have contributed to a reduction in resources needed for the execution of AI algorithms. The development of AI algorithms being implemented on devices has been making consistent progress in recent years. It is now possible to execute AI algorithms on devices rather than cloud servers, and it is anticipated that this trend will continue to grow [1]. Processing that happens locally on the device, as opposed to that on the cloud, and saves time and energy expenditure and enhances data security, since there is no need to transmit raw data between the source device and a distant back-end server.

AI algorithms are built to analyze enormous amounts of data, identify patterns within those data, and then learn from those data. AI algorithms refer to a host of methods used to solve complex problems, recognize patterns, and make predictions based on data. Some examples of well-known AI algorithms include gradient boosting, support vector machines (SVM), random forest, decision trees, k-means, artificial neural networks (ANNs), etc. To develop an AI algorithm, necessary data are first collected and preprocessed. The next step is to train, validate, and test the algorithm. The goal of training is to teach the AI algorithm

to recognize patterns and make accurate predictions or decisions based on the test data it receives. The goal of validation is to check how well the trained model works and optimize it. The goal of testing is to provide an assessment of the final model. The common types of learning approaches for AI algorithms include (1) deep learning, (2) supervised learning, (3) unsupervised learning, and (4) reinforcement learning [2]. In deep learning, large quantities of data are sent into the network (with many layers) during training. Supervised learning occurs when the algorithm is provided with both inputs and outputs, whereas unsupervised learning involves only the provision of inputs. Reinforcement learning, on the other hand, requires the model to learn through experimentation and feedback, as it receives rewards or penalties in response to its actions. Deep neural networks (DNNs) are a popular and effective type of AI algorithm. They have proven to be one of the most efficient methods for dealing with complex data such as videos and images. They consist of numerous layers of processing in between their input and output layers [3]. For DNNs that analyze images and videos, a widely used technique is to combine convolutional filter operations with matrix multiplications. A DNN model contains larger quantities of parameters and calculations, requiring larger memory when compared to other types of ANNs. In addition, a DNN demands a large amount of processing time, energy, and computing resources. Deploying DNNs on resource-constrained devices is thus a difficult problem. Therefore, DNNs are often deployed on remote back-end cloud servers. In some emerging applications, e.g., autonomous vehicles, data need to be analyzed where they are captured by sensors instead of backend servers. Therefore, the execution of a trained DNN model should take place on low-resource devices. This helps circumvent problems associated with latency-sensitive data, security and privacy data, and always-on service. When deploying a DNN model on a resource-constrained device, there is a need to make compromises between several criteria such as the model's size, the amount of energy used, and processing speed [4]. To enable the execution of DNNs on resource-restricted devices, there have been advancements in creating methods based on software, hardware, or combined hardware and software.

On the software side, new programming frameworks and libraries have been created to make it simpler to design DNNs for resource-constrained devices. These include well-known frameworks like TensorFlow, PyTorch, and Keras. These frameworks provide high-level abstractions and pre-built components for DNN development such as training and inference. In addition, there have been developments in the tools that are used for data preparation, model selection and hyperparameter tuning, and model deployment and serving, as well as other applications. On the hardware side, customized processors and architectures have been built specially to boost the deployment and execution of DNNs. These include graphics processing units (GPUs), field-programmable gate arrays (FPGAs), application-specific integrated circuits (ASICs), and tensor processing units (TPUs). There have been a few publications containing a conceptual description of implementing DNNs on Internet of things (IoT) devices with joint back-end computation. These papers, however, do not provide an analysis of the current work in the same field of research. Accordingly, the contributions of this paper include a detailed review of the existing methods and architectures for optimizing DNNs and executing them on resource-constrained devices. This review is presented under three categories: software, hardware, and hardware/software co-design. In addition, a comparison of the methods in the three categories is given together with a discussion on the suitability of the reviewed methods.

2. Overview of DNN

A DNN architecture is constructed by connecting a series of layers, where each layer consists of a grouping of units (neurons) interconnected with one another as shown in Figure 1. In a DNN architecture, the input layer is the initial layer that receives the raw data, which may consist of audio or images. On the other hand, the output layer is the final layer that represents the inferred classes. The remaining layers are the hidden layers which convert the input values into the inferred classes. Every unit has its own activation function,

which defines how to compute the unit's own state. Weights are learnable parameters on the connection lines between the layers converting an incoming value to adjust an outgoing value [5].

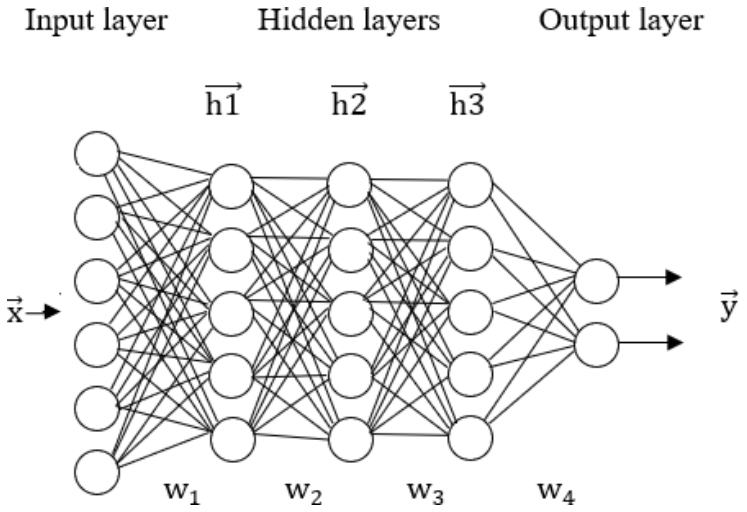

Figure 1. A sample feedforward deep neural network.

There are a variety of DNN architectures. As an example, with a feedforward DNN, the inference starts at the input layer and then proceeds forward layer by layer until it has completed. The output of each unit is updated in a feedforward fashion at each successive layer. When all the units in the output layer have been updated, the classification results will become available. The inferred class is equivalent to the output layer unit that has the most significant value. The majority of DNN's operations consist of layer-wise multiplications and accumulations. All the inputs are multiplied by their respective weights.

Figure 1 depicts a feedforward neural network that consists of an input layer, three hidden layers, and one output layer. Each layer in the network is connected to the next through a weight matrix, resulting in a total of four weight matrices: w_1 w_2, w_3, and w_4. When an input vector $\vec{x}$ is provided, the first weight matrix w_1 is used to calculate a dot product, and the activation function is applied to the result. This produces a new vector $\vec{h_1}$ representing the neurons' values in hidden layer 1. $\vec{h_1}$ is then employed as the input vector of the following hidden layer, repeating the described operations. The same process is repeated to obtain the final output $\vec{y}$ which serves as the networks' prediction. The complete process is expressed through the subsequent equations, in which σ denotes an activation function:

$$\begin{aligned} \vec{h_1} &= \sigma(\vec{x} \cdot w_1) \\ \vec{h_2} &= \sigma(\vec{h_1} \cdot w_2) \\ \vec{h_3} &= \sigma(\vec{h_2} \cdot w_3) \\ \vec{y} &= \sigma(\vec{h_3} \cdot w_4) \end{aligned} \tag{1}$$

3. Implementation of DNN On-Device

In recent years, developments in software, hardware, or combined hardware and software have made it feasible to implement DNNs directly on devices that contain sensors. The execution of data in real-time on the cloud comes with a drawback of increased latency. The communications that take place between devices and a cloud server are long-distance and have a limited bandwidth, which is the cause of latency [1]. Running DNNs on a cloud

server presents additional security and privacy challenges. During communications, there is the possibility that sensitive information may be accessed improperly. The reliability of the communication connection is another challenge that comes with operating DNNs on a cloud server. Safe and secure processing of sensor data by DNNs needs to be established for reliable operation. Thus, implementing real-time data processing directly on the device itself is strongly recommended [6].

To deploy a DNN on a resource-constrained device, a number of approaches can be employed. A DNN typically includes millions of parameters and processes, so several approaches have been developed to optimize the parameters and processes of the DNN while simultaneously maintaining its classification accuracy. Examples of such approaches include network optimization, quantization and compression, data augmentation, power management, and hardware acceleration approaches. These approaches can be categorized into three groups: software, hardware, and hardware software co-design. The software approaches include pruning, compact structure, quantization, and so on. The hardware approaches alter the architecture, design, and memory hierarchy and dataflow mapping. The hardware software co-design includes work carried out based on both hardware and software techniques.

3.1. Software Approaches

In DNNs, compression refers to the process of shrinking the model's size by eliminating redundant or unnecessary parameters without substantially compromising its performance. Several approaches have been developed to compress a DNN and fit it into a resource-constrained hardware platform. These approaches have been illustrated in the following. Pruning refers to the elimination of the connections of a DNN that are not highly essential for its operation. As a result, its memory footprint and computational requirement are reduced to improve its energy efficiency and fit it into a low-resource device [7]. Several pruning techniques have been developed, including weight pruning, neuron pruning, filter pruning, structured pruning, and unstructured pruning. To achieve additional compression, the pruning procedure may be carried out either during or after training. Pruning should be used in combination with other strategies like regularization to preserve the efficiency of the network. If pruning is not performed appropriately, it might also lead to a loss of accuracy.

Weight sharing reduces the memory footprint of the model by allowing multiple neurons to use the same set of weights, which can reduce the number of unique parameters that need to be stored in memory [8]. The common method for achieving weight sharing in CNNs is to apply the same set of weights to each position of the input feature maps. These refer to a 3D matrix that represents the input data for a convolutional layer. The input feature map has a width, a height, and a depth, where the depth corresponds to several channels in the input data.

In a CNN, the same set of weights, also called filters or kernels, is used to convolve the input feature map at different locations, resulting in different output feature maps. Each filter generates one output feature map, and the number of filters determines the number of output feature maps produced by the layer [9]. Each output feature map represents a set of learned spatial features that the layer is sensitive to. Siamese networks typically combine the two identical sub-networks with the same set of weights at some point in the network, either through concatenation or by determining the absolute difference or Euclidean distance between their outputs [10].

Weight tying involves arranging the weights of several layers to be equal in order to bind them together. This is often accomplished by designating a layer as a child layer of another layer, after which the weights of the child layer are adjusted to match those of the parent layer [11]. The idea behind weight tying is to decrease various parameters in the network, which can help prevent overfitting and improve generalization.

Compact structure in DNNs refers to the process of reducing the number of parameters in the network by simplifying its architecture or by using more efficient building blocks.

Methods such as depth-wise separable convolution, squeeze-and-excitation (SE) blocks, ResNet skip connections, MobileNet architecture, and transformer-based architectures help achieve compact structures in DNN. Low-rank approximation involves approximating the weight matrices in a DNN with low-rank matrices, which can reduce the number of parameters and computations required during inference [12]. Some common methods for low-rank approximation in DNNs include singular value decomposition (SVD), tucker decomposition, low-rank factorization, and matrix completion. Low-rank approximation can be used to enhance generalization ability of the network. The specific method used for low-rank approximation depends on the type of network and the specific implementation.

In the technique of knowledge distillation [3], a smaller and less intricate network is trained to replicate the behavior of a larger and more intricate network, which can lead to the development of a condensed model that performs comparably to its larger counterpart. Several techniques used for knowledge distillation in DNNs are temperature scaling, feature maps, inter-layer activations and multi-task learning. Knowledge distillation can be a powerful technique for transferring knowledge to a smaller, simpler model from a larger, more complex model (see Figure 2).

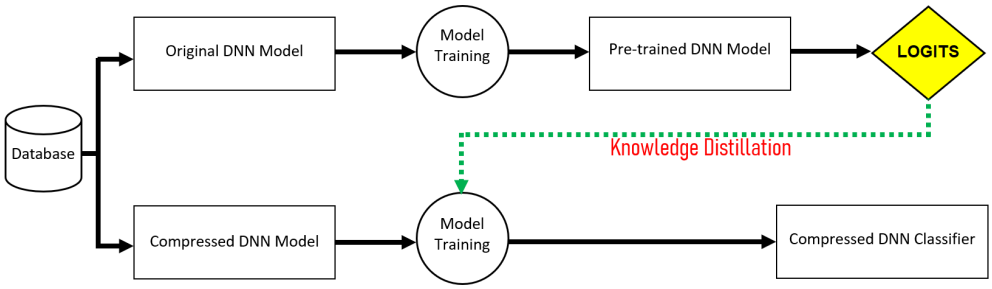

Figure 2. Sketch of knowledge distillation of a DNN model by logits transfer, adopted from [3].

Winograd transformation involves transforming the convolution operation in a CNN into a set of smaller matrix multiplications, and thus can reduce the number of computations required during inference [13]. This can result in a smaller and more efficient model, while maintaining the accuracy of the larger model. Techniques used for implementing the Winograd transformation include F(2×2,3×3) and G(2×2,3×3) transformations, direct convolution, hybrid approaches, and optimization techniques. The specific techniques used for implementing the Winograd transformation depend on the architecture of the CNN, the size of the convolutional kernels, and the available computational resources. The goal is to balance the computational efficiency of the convolution operation with the accuracy and generalization performance of the model.

Quantization is the process of transforming deep learning models to use parameters and computations at a lower precision. By using lower-precision data types, the model's memory requirements can be reduced, which can make it easier to deploy on devices with limited memory. Some types of quantization techniques used in DNNs are fixed-point quantization, dynamic range quantization, hybrid quantization, and vector quantization. The parameters of a DNN may be quantized into predetermined levels through the process of quantization, which results in a reduction in the number of bits that are required to hold those parameters. As is the case with pruning, it requires striking a balance between the two competing goals of reducing the size of the DNN and maintaining its accuracy. At the other end of the spectrum are the binarized DNNs, which have weights and activations that are either +1 or −1 [14]. This results in a reduction in the amount of memory that is required to keep the parameters, and the action of multiplication and adding in binary DNNs is converted into a simple XNOR operation, which is much quicker and saves a significant amount of energy [15].

3.2. Hardware Approaches

For implementing a DNN algorithm on a resource-constrained device, a number of hardware platforms can be used. These include digital signal processors (DSPs), systems-on-chips (SoC), graphics processing units (GPUs), field-programmable gate arrays (FPGAs), and neural processing units (NPUs). Some popular SoC platforms for DNNs include Qualcomm Snapdragon, Apple A-series, Nvidia Jetson, etc. Some popular FPGA platforms for DNNs include Xilinx FPGAs, Intel FPGAs, and Amazon F1 instances.

In addition, microcontrollers have been recently used for hosting DNNs. The following microcontroller platforms are recommended for hosting DNNs [16]: Arduino Nano 33 BLE Sense, STM32 microcontrollers, SparkFun Edge, Espressif ESP32-DevKitC, Adafruit EdgeBadge, Espressif ESP-EYE, Wio Terminal: ATSAMD51, Himax WE-I Plus EVB Endpoint AI Development Board, Synopsys DesignWare ARC EM Software Development Platform, and Sony Spresense.

To deploy an optimized DNN into a resource-constrained device, hardware-based methods have been developed as discussed in the following. When deploying DNNs on resource-constrained devices, it is important to select an appropriate hardware architecture that can support the computational requirements of the DNN while minimizing power consumption and memory usage. There have been attempts made to decrease the total time needed by the network to make an inference on the device. The majority of DNN operations require a lot of processing power, such as multilayered matrix multiplications. However, low-resource devices often have restricted memory size and computational power. As a result, DNN architectures that use much less processing power are becoming more common [7]. Convolution and matrix multiplication operations should be performed using specific hardware blocks, and low-precision data formats should be used for weight, and activation values and the network's size and complexity should be supported by the memory and processing capabilities of the selected platform. Since the energy requirement of low-resource devices needs to be limited, it is essential to optimize the power usage of the DNN execution. Power reduction techniques may be utilized in hardware, such as power gating, dynamic voltage, frequency scaling, and clock gating without compromising performance. Memory is often limited to low-resource devices; hence, memory utilization and optimization are crucial. Without losing speed, hardware modifications like memory compression, caching, and scratchpad memory may be employed to lower the memory needs of DNNs. DNN calculations may be accelerated using on-chip accelerators like DSPs or GPUs. On-chip accelerators may decrease the computational demands of DNNs and enhance performance by offloading computation off the primary processor. The calculations needed by DNNs may be sped-up FPGAs, SoCs, or NPUs. DNN calculations may be accelerated by using parallel processing methods like pipelining or parallel execution. The performance of DNNs on low-resource devices may be enhanced by parallel processing and dividing the calculations into smaller, parallelizable tasks.

Most of the power is consumed during the process of obtaining the parameters and weights from the main memory to the processor, which is where the actual calculations are carried out. The implementation of the DNN requires the use of specialized hardware to be successful. This often requires the expenditure of a substantial quantity of energy [17]. Additionally, the delay rises whenever data are retrieved from the main memory. These issues are being tackled by DNN accelerators. In a broad sense, it is possible to represent the training and inference processes of DNNs as matrix multiplications, unlike those seen in graphics processing. Therefore, GPUs have emerged as the pillar for training and deploying DNNs. The single instruction, multiple threads (SIMT) parallelization approach is used to reduce latency during the DNN inference process. The matrix multiplications may also be optimized for quicker calculations using software tools, which is another way to enhance current hardware [17].

Memory hierarchy in DNNs refers to the organization of different types of memory used in a DNN system to store and access data during training and inference [18]. The primary objective of the memory hierarchy in DNNs is to enhance their efficiency and

performance by utilizing faster, smaller, and costlier memory at the higher levels of the hierarchy, and slower, larger, and more economical memory at the lower levels. Memory hierarchy approaches that can be used to improve the performance of DNNs include caching, tiling, compression, and data layout optimization. Memory hierarchy is created so that data may be delivered rapidly and prevent memory access from being performed twice. The generic memory hierarchy provides support for dataflow mapping and lowering the amount of energy used and enhancing acceleration. Dataflow mapping is the process of mapping the computations involved in DNN onto the available hardware resources. It involves breaking down the DNN computation into smaller blocks and mapping them onto the hardware resources in a way that maximizes the utilization of the available resources and minimizes the overall execution time of the DNN. Some specific techniques used in optimizing dataflow mapping in DNNs are spatial and temporal locality, memory reuse, parallelism, and custom hardware accelerators. These techniques can help optimize the placement and movement of data and computations and improve the overall performance of DNN computations on a wide range of platforms. It is necessary to perform dataflow mapping since DNN parameters may not be stored on the constrained on-chip memory [1]. Thus, an external memory may be used to transmit the parameters.

3.3. Hardware/Software Co-Design Approaches

DNN inference can be deployed on resource-constrained devices using hardware/software co-design approaches. These approaches consist of a combination of a hardware approach and a software approach. Examples of hardware approaches include power reduction techniques; parallel processing methods like pipelining or parallel execution; the single instruction, multiple threads (SIMT) parallelization approach; memory hierarchy; dataflow mapping, etc. Examples of software approaches include pruning, weight sharing, weight tying, compact structure, low-rank approximation, knowledge distillation, Winograd transformation, quantization, etc. As the hardware and software approaches have already been described in the previous subsections, they are not therefore explained here.

4. Existing On-Device DNNs

The works that have already been reported on the implementation of DNNs on resource-constrained devices can be categorized either into software, hardware, and hardware/software co-design approaches. The following subsections provide a review on the reported works involving software, hardware, and hardware/software co-design techniques.

4.1. Current Works Based on Software Approaches

Yao et al. [19] propose a novel approach based on a compressor–critic framework. The compressor-critic framework consists of two components: a compressor network and a critic network. The compressor network takes a pre-trained deep neural network as input and compresses it by removing unnecessary parameters and reducing the number of layers. The critic network evaluates the compressed network and provides feedback to the compressor network, guiding the compression process to produce a compressed network with high accuracy. They also use a reinforcement learning approach to train the compressor and critic networks, with the goal of maximizing the accuracy of the compressed network while minimizing the compression rate. The authors also introduce a novel technique called "layer fusion," which combines multiple layers of the original NN into a particular layer in the compressed network, reducing the number of parameters and improving the compression rate. The paper provides experimental results demonstrating the effectiveness of the proposed method, showing that it can achieve high compression rates while maintaining high accuracy on a range of datasets and deep neural network architectures.

Molchanov et al. [20] propose a method for pruning CNNs by removing unimportant weights. The pruning process is based on the magnitude of the weights in the network, where weights with small magnitudes are considered unimportant and can be pruned without significantly affecting the network's accuracy. Several different pruning tech-

niques are proposed, including pruning individual weights, entire filters, and even entire layers of the network. To mitigate the potential accuracy loss caused by pruning, the authors propose a technique called "iterative pruning and retraining," where the network is repeatedly pruned and then retrained on the pruned architecture to recover the lost accuracy. The authors also propose a method for initializing the pruned network with the weights of the original network, which can speed up the convergence of the retraining process. This paper [20] provides experimental results demonstrating the effectiveness of the proposed methods, showing that it can achieve significant reductions in the number of parameters and computations without sacrificing accuracy on a range of datasets and CNN architectures.

Anwar et al. [21] propose a method for pruning large blocks of filters simultaneously, which can significantly reduce several parameters and computations needed during inference. The method is based on a two-stage process. In the first stage, the network is trained to identify the redundant blocks of filters using a set of heuristics, such as the L1-norm of filter weights or activations. In the second stage, the redundant blocks of filters are pruned, and the network is retrained on the pruned architecture to recover the lost accuracy. To further optimize the network, a technique is proposed called "coarse pruning," where entire blocks of filters are removed from the network, rather than individual filters, which significantly decrease the calculation and memory requirements of the network without sacrificing accuracy. The aforementioned paper [21] provides experimental results demonstrating the effectiveness of the proposed methods, showing that it can achieve significant reductions in the number of parameters and computations without sacrificing accuracy on a range of datasets and CNN architectures.

Yang et al. [22] propose a method for designing energy-efficient CNNs through a technique called energy-aware pruning. The main idea behind energy-aware pruning is to identify and remove the less important connections in a CNN that contribute less to its overall accuracy. By removing these connections, the resulting pruned network has fewer operations to perform, and therefore requires less energy to compute, while maintaining a high level of accuracy. Firstly, a ranking algorithm to identify the most significant connections in the network is used. The ranking algorithm assigns a score to each connection based on its contribution to the overall accuracy of the network. Connections with lower scores are then pruned. Secondly, an iterative pruning approach is used where they gradually prune the network while monitoring its accuracy. At each iteration, a certain percentage of the connections with the lowest scores are pruned and retrained in the network to recover their accuracy. This process is repeated until the required level of pruning is achieved. The proposed pruning technique is applied to several popular CNN architectures, including VGG-16, ResNet-50, and AlexNet, showing that the pruned networks achieve significant energy savings while maintaining a similar level of accuracy compared to the original networks. The authors also compare their method to other pruning techniques and show that their approach outperforms them in terms of energy savings and accuracy. Overall, the paper provides a useful method for designing energy efficient CNNs by leveraging the benefits of pruning techniques.

Narang et al. [23] propose a method for introducing sparsity into the weight matrices of recurrent neural networks (RNNs), which can reduce various computations and memory accesses essential during inference. The sparsity is introduced through a process called "structured pruning," where groups of weights in the weight matrices are pruned simultaneously to maintain the structure of the matrices. The authors propose several different pruning techniques, including pruning individual weights, entire rows and columns, and even entire matrices. To mitigate the potential accuracy loss caused by pruning, the authors propose a technique called "iterative pruning and retraining," like the method proposed in [20], where the network is repeatedly pruned and then retrained on the pruned architecture to recover the lost accuracy. The paper provides experimental results demonstrating the effectiveness of the proposed method, showing that it can achieve significant reductions

in the number of parameters and computations without sacrificing accuracy on a range of datasets and RNN architectures.

Guo et al. [24] propose a method for surgically removing redundant neurons and connections that do not contribute significantly to the network's performance during training, without compromising the network's accuracy. The method is based on a two-stage process. In the first stage, the network is trained to identify the redundant components using a set of heuristics, such as the magnitude of weights or the magnitude of activations. In the second stage, the redundant components are surgically removed, and the network is retrained on the pruned architecture to recover the lost accuracy. To further optimize the network, the authors propose a technique called "channel pruning," where entire channels of feature maps are removed from CNNs, which can significantly reduce various computations and memory accesses essential during inference. The paper provides experimental results demonstrating the effectiveness of the proposed method, showing that it can achieve significant reductions in the number of parameters and computation without sacrificing accuracy on a range of datasets and architectures.

Hinton et al. [25] propose a method called "knowledge distillation", which is for reducing the size and computational complexity of DNNs while maintaining their accuracy. The authors use a smaller, simpler neural network (referred to as the student network) to mimic the behavior of a larger, more complex neural network (referred to as the teacher network) by learning to match the teacher network's output. This is carried out by using the output probabilities of the teacher network as soft targets during the training of the student network. In other words, instead of using the hard labels (e.g., one-hot encoded vectors) typically employed during supervised training, the student network is trained to match the probability distribution produced by the teacher network for each input. In addition to using soft targets, the authors introduce several regularization techniques to prevent overfitting and improve the generalization of the student network. These techniques include adding a temperature parameter to the softmax function used to calculate the output probabilities and adding a term to the loss function that encourages the student network to produce similar activations as the teacher network. The paper provides several experimental results demonstrating the effectiveness of the proposed method, including comparisons with other methods for reducing the size of neural networks. The results show that knowledge distillation can reduce the size of neural networks by a factor of two to ten while maintaining or even improving their accuracy.

Ravi et al. [26] propose a three-stage approach to using deep learning for on-node sensor data analytics. The first stage involves collecting data from the sensors on the device, which can include accelerometers, gyroscopes, magnetometers, or other sensors depending on the application. The collected data are then preprocessed to remove noise, normalize the data, and perform feature extraction to prepare it for input into the deep learning model. In the second stage, a deep learning model is used to process the preprocessed data. It is proposed using a CNN architecture, which is well suited for processing sequential data such as sensor readings. CNN is trained on a labeled dataset of sensor data using supervised learning, where the labels represent the activity or state that corresponds to each sequence of sensor readings. Finally, in the third stage, the trained model is deployed on the device for real-time processing of new sensor data. A sliding window approach is used to process the data in real time, where the window size is determined by the expected duration of the activity or state being classified. The model outputs a prediction for each window of sensor data, which can be used to trigger actions or provide feedback to the user. This approach achieves high accuracy on both datasets while consuming minimal resources, making it well suited for deployment on mobile or wearable devices with limited processing power and battery life.

He et al. [27] propose the ResNet architecture, which introduces residual connections, which allow the network to learn residual functions instead of directly learning the underlying mapping. In the ResNet architecture, each layer learns a residual function that is added to the input of the layer. This allows the network to learn the difference between the input

and the desired output, rather than trying to directly learn the output from the input. This makes it easier for the network to optimize the weights and avoid the vanishing gradient problem. The ResNet architecture also includes bottleneck blocks, which reduce the number of parameters and improve computational efficiency. These blocks consist of three layers, with the first and last layers having fewer filters than the middle layer. This reduces the number of parameters while still allowing the network to learn complex features.

Ham et al. [28] propose the NNStreamer framework, which provides an efficient and flexible pipeline for on-device AI development. The framework includes modular components that can be easily integrated and customized to suit different hardware platforms and use cases. The NNStreamer pipeline includes several components such as data source, preprocessing, neural network, post-processing, and data sink, which can be customized and connected in a flexible manner. The framework also provides a set of built- in operators that can be used for common preprocessing and post-processing tasks, as well as support for hardware acceleration and optimization. The paper provides several experimental results demonstrating the effectiveness and efficiency of the NNStreamer framework, including benchmarks on various hardware platforms and use cases such as object detection and image classification.

4.2. Current Works Based on Hardware Approaches

Shen et al. [29] trained a CNN model using a large dataset of images containing human heads and non-human objects. They used this model to develop an algorithm that can classify whether an image contains a human head or not. The authors then optimized the algorithm for implementation on an edge device by decreasing the number of operations and weights needed for the network to run efficiently on the chip. To implement the algorithm on an edge device, the authors used Mipy evaluation board. "AVSdsp", a technology company, created the Mipy evaluation board to execute the task of detecting human heads through images. They devised a software pipeline that comprises the subsequent stages. Firstly, an image is captured using a camera connected to the FPGA board. Secondly, the image is preprocessed to resize it to the desired input size for the CNN model. Thirdly, the image is run through the CNN model on the FPGA to detect whether a human head is present. Finally, the results of the human head detection algorithm are displayed on an output display. The authors evaluated the performance of their implementation by measuring the power consumption, accuracy, and real-time performance of the algorithm on the FPGA board. They showed that the implementation achieves high accuracy and real-time performance while consuming minimal power, making it suitable for use in edge AI applications.

Dong et al. [30] propose a system that employs a hierarchical architecture consisting of edge devices and cloud servers. The edge devices capture video streams and perform preliminary processing to decrease the amount of data sent to the cloud. The cloud servers are responsible for performing more complex processing, such as object detection and tracking, using deep learning models. They describe the hardware implementation of their system using Nvidia Jetson TX2 edge devices. They also propose a data compression technique to reduce the amount of data sent from the edge devices to the cloud, thereby reducing latency and bandwidth requirements. The authors evaluate their system on an open-air object detection dataset and show that their system achieves real-time performance with high accuracy. They also compare their system to existing state-of-the-art systems and show that their system outperforms them in terms of accuracy and processing speed.

Suleiman et al. [31] propose the concept of visual–inertial odometry (VIO), which uses a custom CNN architecture, which is a navigation method that combines visual and inertial data to estimate the position and orientation of a drone in real time. They then discuss the challenges associated with implementing VIO on small drones that have limited power, processing, and memory resources. The paper presents Navion, a fully integrated hardware accelerator that is designed to be low-power and lightweight, making it ideal for use on small drones. The accelerator is based on a custom-designed systolic

array architecture that is optimized for VIO computations. The authors also present an algorithm that is specifically designed to run on Navion, which combines feature tracking and filtering techniques to estimate the drone's position and orientation. The authors evaluate the performance of Navion using a custom-built quadcopter and compare it with other state-of-the-art VIO algorithms. They demonstrate that Navion can achieve accurate and real-time navigation while consuming only 2 milliwatts of power, which is several orders of magnitude lower than other VIO accelerators.

Li et al. [32] propose a hardware architecture to implement the accelerator, which consists of multiple processing elements (PEs) connected through a network on chip (NoC) to form a systolic array. The PEs perform the convolution and pooling operations in parallel, which allows for high throughput and low latency. The authors also developed a software stack that includes a compiler and a runtime system to translate CNN models into configurations that can be executed on the FPGA accelerator. The compiler optimizes the CNN model by mapping its layers to the PEs and generating configuration files for the FPGA. The runtime system manages the communication between the host computer and the FPGA and controls the execution of the CNN model. The authors evaluated the performance of their FPGA-based accelerator by comparing it with a software implementation of the same CNN model running on a high-end CPU. They showed that their FPGA-based accelerator achieved significant speedup and energy efficiency compared to the CPU implementation, while maintaining high accuracy.

Dinelli et al. [33] use a CNN architecture called SqueezeNet as the target network for their accelerator. The hardware accelerator is designed to perform the convolutional and pooling operations of the SqueezeNet architecture using only on-chip memories, which helps to reduce the latency and energy consumption of the accelerator. The hardware accelerator is implemented on a Xilinx Zynq UltraScale+ FPGA, which includes a processing system and programmable logic. The authors use programmable logic to implement the hardware accelerator, which consists of a convolutional layer, a max-pooling layer, and a ReLU activation layer. The on-chip memories are used to store the weights and input feature maps of the convolutional layer and the output feature maps of the max-pooling and ReLU layers. The authors benchmark their hardware accelerator using the Intel Movidius Neural Compute Stick, a popular edge device for running neural networks. They compare the performance of their hardware accelerator with the Neural Compute Stick using two different CNN architectures, SqueezeNet and MobileNet. The results show that the hardware accelerator outperforms the Neural Compute Stick in terms of both latency and energy consumption for both CNN architectures.

Chen et al. [34] propose an energy-efficient accelerator for CNNs called Eyeriss. Eyeriss is designed to optimize the computation and communication patterns of CNNs, which are the most computationally intensive operations in deep learning. Eyeriss uses a reconfigurable architecture that allows it to adapt to different CNN models and layers. The architecture is composed of PEs that perform the convolution and pooling operations and a memory hierarchy that stores the weights, activations, and feature maps. The PEs are arranged in a mesh topology, and each PE is connected to its neighboring PEs through a communication network. Eyeriss also uses a dataflow organization that optimizes the communication between the PEs and the memory hierarchy. The dataflow organization ensures that the data are streamed from memory to the PEs only when they are needed, and that the data are processed in the correct order. This reduces data movement and energy consumption. To further reduce energy consumption, Eyeriss uses a technique called weight stationary, which exploits the fact that the weights in a CNN are reused multiple times. The weight stationary technique stores the weights in a separate memory bank and streams them to the PEs as needed, reducing the energy consumption of fetching the weights from memory. The authors of the paper implemented Eyeriss on a FPGA platform and evaluated its performance on several CNN models. The results show that Eyeriss achieves high energy efficiency compared to other state-of-the-art accelerators while maintaining high accuracy on image classification tasks.

Chen et al. [35] propose the Eyeriss v2 accelerator, which is implemented using a tiled architecture, where each tile consists of a PE array and a local memory. The processing element array is responsible for performing the computation of the neural network layers, while the local memory stores the weights and input feature maps of the layers. The Eyeriss v2 accelerator uses a flexible dataflow scheduling scheme to handle the complex computation and data movement patterns of emerging deep neural networks. The dataflow scheduling scheme allows the accelerator to dynamically allocate resources to different layers of the network, based on their computation and memory requirements. The authors use several emerging architectures, such as RNNs and attention-based models, to evaluate the performance of the Eyeriss v2 accelerator. The results show that the Eyeriss v2 accelerator achieves high energy efficiency and performance compared to state-of-the-art mobile accelerators, such as the Google Pixel Visual Core and the Apple Neural Engine. The authors also introduce a new performance metric called the "energy- delay product" (EDP), which considers both the energy consumption and the latency of the accelerator. The EDP metric shows that the Eyeriss v2 accelerator outperforms other mobile accelerators in terms of energy efficiency and latency.

4.3. Current Works Based on Hardware/Software Co-Design Approaches

Lane et al. [5] propose a system called DeepX, which accelerates the inference of DNNs on mobile devices. The authors use a combination of software techniques and hardware optimizations to achieve this goal (see Figure 3). The software techniques include optimizing the computation graph, pruning the network, and quantizing the weights and activations. The hardware optimizations include using a custom hardware accelerator and reducing the number of memory accesses. The authors use a custom instruction set architecture (ISA) to efficiently map the computations required for deep learning inference onto the hardware accelerator. The hardware accelerator is designed to work in conjunction with a software stack that includes a compiler, runtime library, and driver. The authors evaluate their system on several benchmark datasets and compare it to existing state-of-the-art systems. They show that their system achieves a significant speedup compared to the baseline system, while consuming significantly less power. They also show that their system achieves state-of-the-art performance on several benchmark datasets. Overall, the paper presents an interesting approach to accelerating deep learning inference on mobile devices. The authors provide a detailed description of their system and evaluation methodology, and the results show that their system is effective in reducing the power consumption of deep learning inference on mobile devices while maintaining high accuracy. However, it should be noted that the proposed system is still in the research stage and has not been widely deployed or evaluated in real-world scenarios.

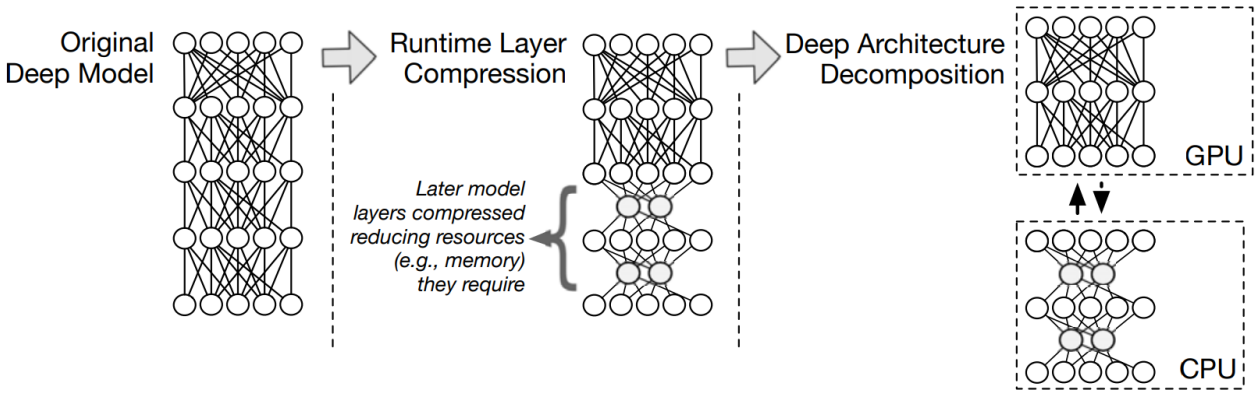

Figure 3. Model decomposition and compression in DeepX; reprinted with permission from Ref. [5].

Ding et al. [36] propose a novel method for quantizing weights and activations of DNNs using integer and binary representations, which reduces the amount of memory required to store the network parameters and improve the computational efficiency of the

inference process. They show that their proposed method achieves competitive accuracy on benchmark datasets such as CIFAR-10 and ImageNet, while reducing the memory footprint and energy consumption by up to 4× compared to full-precision networks. The authors also propose a hardware architecture for implementing the quantized DNN, which uses parallel processing and pipelining to accelerate the inference process. The hardware accelerator is designed to support low-precision computations, specifically 1-bit and 2-bit quantization. This enables the accelerator to perform computations using a reduced number of bits, which reduces power consumption and hardware complexity. To evaluate their implementation, the authors tested their system on several benchmark datasets, including CIFAR-10 and SVHN. The results showed that their hardware accelerator achieved high accuracy while consuming significantly less power than existing DNN accelerators.

Zhang et al. [37] propose FitNN, a low-resource FPGA- based CNN accelerator that can be integrated with a drone's on-board processor. This involves the design and implementation of a CNN accelerator software stack for drones, which is optimized for use with a low-resource FPGA-based hardware accelerator. The software stack consists of several components, including a front end that handles input data processing and communication with the FPGA accelerator, and a back end that performs output processing and sends results back to the drone's main processor. The FPGA accelerator itself is designed to perform the computationally intensive convolutional and pooling operations that are required for CNN inference. The authors also develop a custom hardware-software co-design methodology that allows for efficient communication between the FPGA accelerator and the software stack. This methodology involves the use of specialized hardware interfaces and a communication protocol that optimizes the transfer of data between the FPGA and the software stack. To evaluate their software and hardware co-design approach, the authors implemented their system on a low-cost FPGA development board and tested it using the CIFAR-10 dataset. The results showed that their approach achieved high accuracy and speed, while consuming very little power.

5. Analysis and Comparison

5.1. Comparison of Software-Based Approaches

The software-based approaches utilize algorithms to implement DNNs on low-resource devices, which can improve efficiency and accuracy. With the wide range of existing works reported in this paper, it is essential to compare their features, benefits, and limitations. In Tables 1 and 2, we compare different software-based approaches to understand their key features and differences, and ultimately make an informed choice for specific needs at hand.

Table 1. Comparison of software-based approaches.

Author	Architecture	Compression Rate	Memory
Yao et al. [19]	DNN	Not Available	Before: 90% After: 98.9%
Molchanov et al. [20]	CNN	Before: 9× After: 13×	Not Available
Anwar et al. [21]	Deep CNN	Not Available	Before: Not Available After: 10×
Yang et al. [22]	CNN	Before: Not Available After: 13×	Not Available
Narang et al. [23]	RNN	Before: Not Available After: 16×	Before: Not Available After: 90%
Guo et al. [24]	DNN	Before: Not Available After: 17.7×	Before: Not Available After: 108×
Hinton et al. [25]	DNN	Not Available	Before: Not Available After: 30×
Ravi et al. [26]	CNN	Not Available	Not Available
He et al. [27]	DNN	Not Available	Before: Not Available After: 52 MB

Table 2. Comparison of software-based features.

Author	Speed	Power Consumption	Compression Ratio	Accuracy
Yao et al. [19]	Not Available	Before: Not Available After: 90.6%	Before: 71.4% After: 94.5%	Not Available
Molchanov et al. [20]	Before: 2× After: 4×	Not Available	Not Available	Before: 2× After: 4×
Anwar et al. [21]	Not Available	Not Available	Before: 98.88% After: 98.78%	Not Available
Yang et al. [22]	Before: Not Available After: 3.7×	Before: Not Available After: 13×	Before: Not Available After: 1%	Not Available
Narang et al. [23]	Not Available	Not Available	Not Available	Before: 2× After: 7×
Guo et al. [24]	Not Available	Not Available	Not Available	Not Available
Hinton et al. [25]	Not Available	Not Available	Before: Not Available After: 80%	Not Available
Ravi et al. [26]	Not Available	Not Available	Not Available	Not Available
He et al. [27]	Not Available	Not Available	Before: Not Available After 3.57%.	Before: Not Available After: 34%

Various techniques such as pruning, compression, and knowledge distillation can significantly reduce the model size and computational requirements of DNNs while maintaining high accuracy on various tasks. In terms of compression rate, Guo et al. [24] reported a greater compression rate of 17.7× when compared with reported works of Molchanov et al. [20], Yang et al. [22], and Narang et al. [23]. In terms of power consumption, the approaches proposed in these papers vary in their focus and methodology, but they all aim to reduce the power consumption of DNNs in different ways. Some papers focus on model compression and pruning, while others focus on design. Yao et al. [19] reported average power consumption of 94.5%, as many papers did not report power consumption of their works. Overall, it is difficult to compare the accuracy results across these papers as they use different datasets and tasks, and some of them focus on aspects other than accuracy. However, Molchanov et al. [20] reported a greater accuracy of 94.2% when compared with other reported existing works' accuracy.

5.2. Comparison of Hardware-Based Approaches

The hardware- based approaches utilize various hardware components such as processors, memory, etc., to fit DNNs on low-resource devices, which can improve efficiency and accuracy. With various existing works we have reported in this paper, it is essential to compare various aspects of these approaches. In Tables 3 and 4, we compare different hardware-based approaches and features to provide greater insight of these reported works.

These papers propose different hardware architectures for accelerating DNNs, with a focus on different application scenarios and optimization goals. Li et al. [32] and Chen et al. [34,35] present designs that achieve high energy efficiency while maintaining accuracy, while Dinelli et al. [33] propose a design that reduces off-chip memory accesses. Shen et al. [29] focus on the application scenario of edge devices, while Chen et al. [34,35] present flexible architectures that support emerging NN models. Suleiman et al. [31] reported greater speed with 200 FPSwhen compared with Dong et al. [30] and Chen et al. [34]. Suleiman et al. [31] reported the lowest power consumption, with 2 mW. Shen et al. [29] reported a higher accuracy of 98.7% when compared with other accuracies reported.

Table 3. Comparison of hardware-based approaches.

Author	Network Type	Accelerator	Platform	Size
Shen et al. [29]	CNN	NA	SoC	Not Available
Dong et al. [30]	CNN	Nvidia Jetson TX2	Not Available	Not Available
Suleiman et al. [31]	CNN	VIO accelerator	CMOS	Before: Not Available After: 6.4 mm
Li et al. [32]	CNN	Xilinx VC709	FPGA	Before: Not Available After: 1.45
Dinelli et al. [33]	CNN	Xilinx, Intel	FPGA	Not Available
Chen et al. [34]	CNN	Eyeriss	NOC	Before: Not Available After: 168 PEs
Chen et al. [35]	DNN	Eyeriss v2	NoC	Before: Not Available After: 1024 PEs

Table 4. Comparison of hardware-based features.

Author	Speed	Power Consumption	Accuracy
Shen et al. [29]	Not Available	Not Available	Before: Not Available After: 98.7%
Dong et al. [30]	Before: Not Available After: 30 FPS	Not Available	Not Available
Suleiman et al. [31]	Before: Not Available After: 200 FPS	2 mW	Before: Not Available After: 2.3%
Li et al. [32]	Before: Not Available After: 156 MHz	Before: Not Available After: 330 W	Not Available
Dinelli et al. [33]	Not Available	Before: Not Available After: 2.259	Before: Not Available After: 90.23%
Chen et al. [34]	Before: Not Available After: 35 FPS	Before: Not Available After: 45%	Not Available
Chen et al. [35]	Before: Not Available After: 42.5×	Before: Not Available After: 11.3×	Before: Not Available After: 80.43%

5.3. Comparison of Hardware-/Software-Based Approaches

Hardware/software co-design approaches leverage the strengths of both hardware and software methods, combining their capabilities to achieve optimized solutions for complex computing tasks. In Tables 5 and 6, we compare various aspects of hardware/software co-design approaches and features. By providing a comprehensive overview of these approaches, we aim to assist in making informed decisions when choosing and implementing hardware/software co-design approaches.

The three listed papers focus on developing efficient accelerators for deep learning inference on different platforms. In terms of memory, Lane et al. [5] reduced the memory usage size by 75.5%, whereas the remaining two papers did not mention the memory usage. In terms of power consumption, Ding et al. [36] reduced the power usage by 61.6%, whereas Zhang et al. [37] reduced the power usage by 1.97 w. Lane et al. [5] improved the speed by 5.8 times when compared with other networks. Overall, it is difficult to compare the accuracy results across these papers, as some papers did not report the accuracy of the developed approaches.

Table 5. Comparison of hardware/software co-design-based approach.

Author	Network Type	Accelerator	Platform	Memory
Lane et al. [35]	DNN	Deepx	SoC	Not Available
Ding et al. [36]	DNN	Not Available	Custom	Not Available
Zhang et al. [37]	CNN	FitNN	FPGA	Not Available

Table 6. Comparison of hardware/software co-design-based features.

Author	Speed	Power Consumption	Accuracy
Lane et al. [35]	Before: Not Available After: 5.8 faster	Not Available	Before: Not Available After: 15%
Ding et al. [36]	Not Available	Before: Not Available After: 61.6%	Not Available
Zhang et al. [37]	Before: Not Available After: 9 FPS	Before: Not Available After: 1.97 W	Not Available

6. Discussions

This review paper examined various hardware/software-based solutions for the implementation of deep learning networks (DNNs) on devices that are limited in resources. These solutions include model compression, quantization, and hardware-based solutions. Model compression allows for a reduction in model size, a decrease in memory consumption, and an increase in inference speed. Quantization techniques reduce the number of parameters, allowing DNNs to be deployed on devices with fewer resources. Additionally, quantization methods can be used to reduce the precision of data types, further reducing memory usage and computing requirements. Finally, hardware-based solutions allow for the offloading of intensive calculations from the primary processor, allowing for more system resources to be used for other tasks. Hardware/software co-design is a combination of hardware and software approaches that seeks to achieve optimal DNN performance on devices with limited resources. By combining hardware accelerators with software optimization, co-designed systems can be more efficient and cost-effective than pure software implementations. This type of approach allows for more granular optimization, including hardware-specific optimization for critical operations, while also taking advantage of software-based modeling and quantization techniques. Both co-design and hardware-based approaches have their own trade-offs, with software-based approaches typically requiring additional hardware components that can increase system complexity and cost. Ultimately, the selection of an approach is dependent on the device's resource limitations, performance needs, and the desired application. Table 7 provides a comparison of all three approaches for various parameters that were discussed earlier.

Table 7. Comparison of all three approaches.

Parameter	Approach	Author	Network Type	Accelerator	Obtained Result
Power consumption	Software	Yao et al. [19]	DNN	Not available	90.6%
	Hardware	Suleiman et al. [31]	CNN	VIO accelerator	2 mW
	Hardware/ software co-design	Zhang et al. [37]	CNN	FitNN	1.97 W
Compression rate	Software	Guo et al. [24]	DNN	Not available	17.7×
	Hardware	Not available	Not available	Not available	Not available
	Hardware/ software co-design	Not available	Not available	Not available	Not available
Speed	Software	Molchanov et al. [20]	CNN	Not available	4×
	Hardware	Suleiman et al. [31]	CNN	VIO accelerator	200 FPS
	Hardware/ software co-design	Zhang et al. [37]	CNN	FitNN	9 FPS

Table 7. *Cont.*

Parameter	Approach	Author	Network Type	Accelerator	Obtained Result
Memory	Software	Yao et al. [19]	DNN	Not available	98.8%
	Hardware	Not available	Not available	Not available	Not available
	Hardware/ software co-design	Not available	Not available	Not available	Not available
Accuracy	Software	Molchanov et al. [20]	CNN	Not available	94.2%
	Hardware	Shen et al. [29]	CNN	Not available	98.7%
	Hardware/ software co-design	Lane et al. [35]	DNN	DeepX	15% increase

7. Future Prospectus, Challenges, Advantages, Applications

Future Prospectus: Low-resource devices are poised to have a significant impact on various industries and domains in the coming years. With advancements in edge computing and the proliferation of IoT devices, low-resource devices are expected to play a major role in enabling smart cities, smart homes, and smart industries [38]. They can enable real-time data processing, improve automation, and facilitate seamless communication between devices, paving the way for a more interconnected and intelligent world.

Security Considerations: Security is a crucial aspect of low-resource devices, as they are often connected to the Internet and may handle sensitive data. Securing low-resource devices presents unique challenges due to their limited processing power and memory, making it essential to implement efficient security measures [39]. Ensuring data integrity, authentication, and confidentiality are key considerations for securing low-resource devices and protecting against potential cyber threats.

Challenges: Low-resource devices face several challenges, including limited processing power, memory, and storage, which can impact their performance and functionality. Developing NNs that can run efficiently on resource-constrained devices can be challenging, as they require optimization to operate with limited resources [40]. Power management is another challenge, as low-resource devices are often battery-powered and need to be energy-efficient to extend their operational lifespan.

Advantages: Despite their limitations, low-resource devices offer several advantages. They are cost-effective, as they are often cheaper to manufacture and maintain compared to high-end computing devices. Low-resource devices are also compact and portable, making them suitable for deployment in remote or constrained environments [39]. They can enable real-time data processing at the edge, reducing the requirement for data transmission to the cloud and minimizing latency. Additionally, low-resource devices can operate in harsh environments and have longer battery life, making them ideal for IoT and embedded applications.

Applications: Low-resource devices find applications in various domains, including healthcare, agriculture, logistics, industrial automation, and smart home systems [41]. In healthcare, low-resource devices can be used for remote patient monitoring, telemedicine, and wearable health devices. In agriculture, they can enable precision farming, smart irrigation, and livestock monitoring. In logistics, low-resource devices can be utilized for tracking and monitoring assets, optimizing supply chain operations, and improving fleet management. In industrial automation, they can be used for machine monitoring, predictive maintenance, and process optimization [42]. In smart home systems, low-resource devices can control and automate household appliances, lighting, and security systems.

8. Conclusions

Deploying neural networks on low-resource devices can aid in reducing network congestion by enabling computations to be executed near the data sources, maintaining privacy, and lowering power consumption. This study was undertaken to provide a description of the current methods that ensure the performance of neural network models

on hardware with low resources. This review focuses on previously published studies related to the implementation of DNNs on various devices. The examined methods detail how to reduce DNN models and implement them on resource-constrained hardware platforms. This deployment process may be carried out using either software, hardware, or combined hardware and software. It is essential to note that the three approaches are not mutually exclusive. Requirements such as computational complexity, privacy, and energy consumption will determine which software and hardware approach will be required. Software and hardware advancements have played a crucial role in the recent success and broad acceptance of AI technologies, making it faster and more efficient to design and deploy intelligent systems. However, many obstacles remain, such as the need for more efficient and scalable electronic technology, improved algorithms and architectures, and more resilient AI solutions.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Fowers, J.; Ovtcharov, K.; Papamichael, M.; Massengill, T.; Liu, M.; Lo, D.; Alkalay, S.; Haselman, M.; Adams, L.; Ghandi, M.; et al. A configurable cloud-scale DNN processor for real-time AI. In Proceedings of the 45th Annual International Symposium on Computer Architecture, Los Angeles, CA, USA, 1–6 June 2018.
2. Merenda, M.; Porcaro, C.; Iero, D. Edge machine learning for ai-enabled IoT devices: A review. *Sensors* **2020**, *20*, 2533. [CrossRef]
3. Mishra, R.; Gupta, H.P.; Dutta, T. A survey on deep neural network compression: Challenges, overview, and solutions. *arXiv* **2020**, arXiv:2010.03954.
4. Zhichao, Z.; Abbas, Z.K. Implementation of DNNs on IoT devices. *Neural Comput. Appl.* **2020**, *1*, 1327–1356.
5. Lane, N.D.; Bhattacharya, S.; Georgiev, P.; Forlivesi, C.; Jiao, L.; Qendro, L.; Kawsar, F. DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices. In Proceedings of the 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), Vienna, Austria, 11–14 April 2016; pp. 1–12.
6. Wu, B.; Wan, A.; Iandola, F.; Jin, P.H.; Keutzer, K. SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous Driving. In Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), Honolulu, HI, USA, 21–26 July 2017; pp. 446–454.
7. Ramasubramanian, A.K.; Mathew, R.; Preet, I.; Papakostas, N. Review and application of Edge AI solutions for mobile collaborative robotic platforms. *Procedia CIRP* **2022**, *107*, 1083–1088. [CrossRef]
8. Dupuis, E.; Novo, D.; O'Connor, I.; Bosio, A. Fast exploration of weight sharing opportunities for CNN compression. *arXiv* **2021**, arXiv:2102.01345.
9. Chmiel, B.; Baskin, C.; Banner, R.; Zheltonozhskii, E.; Yermolin, Y.; Karbachevsky, A.; Bronstein, A.M.; Mendelson, A. Feature map transform coding for energy-efficient cnn inference. In Proceedings of the 2020 International Joint Conference on Neural Networks (IJCNN), Glasgow, UK, 19–24 July 2020; pp. 1–9.
10. Roy, S.K.; Harandi, M.; Nock, R.; Hartley, R. Siamese networks: The tale of two manifolds. In Proceedings of the IEEE/CVF International Conference on Computer Vision, Seoul, Republic of Korea, 27 October–2 November 2019; pp. 3046–3055.
11. Diaconu, N.; Worrall, D. Learning to convolve: A generalized weight-tying approach. In Proceedings of the International Conference on Machine Learning, Long Beach, CA, USA, 9–15 June 2019; pp. 1586–1595.
12. Jain, S.; Hamidi-Rad, S.; Racapé, F. Low rank based end-to-end deep neural network compression. In Proceedings of the 2021 Data Compression Conference (DCC), Snowbird, UH, USA, 23–26 March 2021; pp. 233–242.
13. Zhang, Q.; Cheng, X.; Chen, Y.; Rao, Z. Quantifying the knowledge in a DNN to explain knowledge distillation for classification. *IEEE Trans. Pattern Anal. Mach. Intell.* **2022**, *45*, 5099–5113. [CrossRef] [PubMed]
14. Courbariaux, M.; Hubara, I.; Soudry, D.; El-Yaniv, R.; Bengio, Y. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or −1. *arXiv* **2016**, arXiv:1602.02830.
15. Rastegari, M.; Ordonez, V.; Redmon, J.; Farhadi, A. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In Proceedings of the Computer Vision–ECCV 2016, 14th European Conference, Amsterdam, The Netherlands, 11–14 October 2016; Volume 9908, pp. 525–542.
16. C.C.A. License. TensorFlow Lite for Microcontrollers. Available online: https://www.tensorflow.org/lite/microcontrollers (accessed on 22 July 2022).
17. Sze, V.; Chen, Y.-H.; Yang, T.-J.; Emer, J.S. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* **2017**, *105*, 2295–2329. [CrossRef]
18. Parashar, A.; Raina, P.; Shao, Y.S.; Chen, Y.H.; Ying, V.A.; Mukkara, A.; Venkatesan, R.; Khailany, B.; Keckler, S.W.; Emer, J. Timeloop: A systematic approach to dnn accelerator evaluation. In Proceedings of the 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Madison, WI, USA, 24–26 March 2019; pp. 304–315.

19. Yao, S.; Zhao, Y.; Zhang, A.; Su, L.; Abdelzaher, T.F. DeepIoT: Compressing Deep Neural Network Structures for Sensing Systems with a Compressor-Critic Framework. In Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems, Delft, The Netherlands, 6–8 November 2017; pp. 1–14.
20. Molchanov, P.; Tyree, S.; Karras, T.; Aila, T.; Kautz, J. Pruning Convo-lutional Neural Networks for Resource Efficient Inference. In Proceedings of the International Conference on Learning Representation (ICLR), Toulon, France, 24–26 April 2017.
21. Anwar, S.; Sung, W. Compact Deep Convolutional Neural Networks with Coarse Pruning. *arXiv* **2016**, arXiv:1610.09639.
22. Yang, T.-J.; Chen, Y.-H.; Sze, V. Designing Energy-Efficient Convo- lutional Neural Networks Using Energy-Aware Pruning. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, 18–20 June 2017; pp. 5687–5695.
23. Narang, S.; Diamos, G.; Sengupta, S.; Elsen, E. Exploring Sparsity in Recurrent Neural Networks. In Proceedings of the International Conference on Learning Representations, Toulon, France, 24–26 April 2017.
24. Guo, Y.; Yao, A.; Chen, Y. Dynamic Network Surgery for Efficient DNNs. In Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16), Barcelona, Spain, 5 December 2016; pp. 1387–1395.
25. Hinton, G.; Vinyals, O.; Dean, J. Distilling the Knowledge in a Neural Network. In Proceedings of the NIPS Deep Learning and Representation Learning Workshop, Montréal, QC, Canada, 11 December 2015.
26. Ravi, D.; Wong, C.; Lo, B.; Yang, G.-Z. A Deep Learning Approach to on-Node Sensor Data Analytics for Mobile or Wearable Devices. *IEEE J. Biomed. Health Inform.* **2017**, *21*, 56–64. [CrossRef] [PubMed]
27. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep Residual Learning for Image Recognition. In Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 27–30 June 2016; pp. 770–778.
28. Ham, M.; Moon, J.; Lim, G.; Jung, J.; Ahn, H.; Song, W.; Woo, S.; Kapoor, P.; Chae, D.; Jang, G.; et al. NNStreamer: Efficient and Agile Development of On- Device AI Systems. In Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Madrid, Spain, 25–28 May 2021; pp. 198–207.
29. Shen, F.-J.; Chen, J.-H.; Wang, W.-Y.; Tsai, D.-L.; Shen, L.-C.; Tseng, C.-T. A CNN-Based Human Head Detection Algorithm Implemented on Edge AI Chip. In Proceedings of the 2020 International Conference on System Science and Engineering (ICSSE), Kagawa, Japan, 31 August–3 September 2020; pp. 1–5.
30. Dong, L.; Yang, Z.; Cai, X.; Zhao, Y.; Ma, Q.; Miao, X. WAVE: Edge-Device Cooperated Real-time Object Detection for Open-air Applications. *IEEE Trans. Mob. Comput.* **2022**, *22*, 4347–4357. [CrossRef]
31. Suleiman, A.; Zhang, Z.; Carlone, L.; Karaman, S.; Sze, V. Navion: A 2-mW Fully Integrated Real-Time Visual-Inertial Odometry Accelerator for Autonomous Navigation of Nano Drones. *IEEE J. Solid-State Circuits* **2019**, *54*, 1106–1119. [CrossRef]
32. Li, H.; Fan, X.; Jiao, L.; Cao, W.; Zhou, X.; Wang, L. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In Proceedings of the 26th International Conference on Field Programmable Logic and Applications (FPL), Lausanne, France, 29 August–2 September 2016; pp. 1–9.
33. Dinelli, G.; Meoni, G.; Rapuano, E.; Benelli, G.; Fanucci, L. An FPGA-Based Hardware Accelerator for CNNs Using On-Chip Memories Only: Design and Benchmarking with Intel Movidius Neural Compute Stick. *Int. J. Reconfigurable Comput.* **2019**, *2019*, 7218758. [CrossRef]
34. Chen, Y.-H.; Krishna, T.; Emer, J.S.; Sze, V. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE J. Solid-State Circuits* **2017**, *52*, 127–138. [CrossRef]
35. Chen, Y.-H.; Yang, T.-J.; Emer, J.S.; Sze, V. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE J. Emerg. Sel. Top. Circuits Syst.* **2019**, *9*, 292–308. [CrossRef]
36. Ding, R.; Liu, Z.; Blanton, R.D.S.; Marculescu, D. Quantized Deep Neural Networks for Energy Efficient Hardware-based Inference. In Proceedings of the 2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC), Jeju, Republic of Korea, 22–25 January 2018; pp. 1–8.
37. Zhang, Z.; Mahmud, M.P.; Kouzani, A.Z. FitNN: A Low-Resource FPGA-Based CNN Accelerator for Drones. *IEEE Internet Things J.* **2022**, *9*, 21357–21369. [CrossRef]
38. Sarker, I.H. AI-Based Modeling: Techniques, Applications and Research Issues. *SN Comput. Sci.* **2022**, *3*, 158. [CrossRef] [PubMed]
39. Tan, B.; Karri, R. Challenges and New Directions for AI and Hard-ware Security. In Proceedings of the 2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS), Springfield, MA, USA, 9–12 August 2020; pp. 277–280.
40. Pabby, G.; Kumar, N. A Review on Artificial Intelligence, Challenges Involved and Its Applications. *Int. J. Adv. Res. Comput. Eng. Technol.* **2017**, *6*.
41. Bezboruah, T.; Bora, A. Artificial intelligence: The technology, challenges and applications. *Trans. Mach. Learn. Artif. Intell.* **2020**, *8*, 44–51.
42. Hu, Y.; Li, W.; Wright, D.; Aydin, O.; Wilson, D.; Maher, O.; Raad, M. Artificial Intelligence Approaches. The Geographic Information Science and Technology Body of Knowledge (3rd Quarter 2019 Edition). *arXiv* **2019**, arXiv:1908.10345. [CrossRef]