Efficient Processing of Deep Neural Networks: A Tutorial and Survey

Vivienne Sze, *Senior Member, IEEE,* Yu-Hsin Chen, *Student Member, IEEE,* Tien-Ju Yang, *Student Member, IEEE,* Joel Emer, *Fellow, IEEE*

***Abstract*—Deep neural networks (DNNs) are currently widely used for many artificial intelligence (AI) applications including computer vision, speech recognition, and robotics. While DNNs deliver state-of-the-art accuracy on many AI tasks, it comes at the cost of high computational complexity. Accordingly, techniques that enable efficient processing of DNNs to improve *energy efficiency* and *throughput* without sacrificing application accuracy or increasing hardware cost are critical to the wide deployment of DNNs in AI systems.**

This article aims to provide a comprehensive tutorial and survey about the recent advances towards the goal of enabling efficient processing of DNNs. Specifically, it will provide an overview of DNNs, discuss various hardware platforms and architectures that support DNNs, and highlight key trends in reducing the computation cost of DNNs either solely via hardware design changes or via joint hardware design and DNN algorithm changes. It will also summarize various development resources that enable researchers and practitioners to quickly get started in this field, and highlight important benchmarking metrics and design considerations that should be used for evaluating the rapidly growing number of DNN hardware designs, optionally including algorithmic co-designs, being proposed in academia and industry.

The reader will take away the following concepts from this article: understand the key design considerations for DNNs; be able to evaluate different DNN hardware implementations with benchmarks and comparison metrics; understand the trade-offs between various hardware architectures and platforms; be able to evaluate the utility of various DNN design techniques for efficient processing; and understand recent implementation trends and opportunities.

I. Introduction

Deep neural networks (DNNs) are currently the foundation for many modern artificial intelligence (AI) applications [1]. Since the breakthrough application of DNNs to speech recognition [2] and image recognition [3], the number of applications that use DNNs has exploded. These DNNs are employed in a myriad of applications from self-driving cars [4], to detecting cancer [5] to playing complex games [6]. In many of these domains, DNNs are now able to exceed human accuracy. The superior performance of DNNs comes from its ability to extract high-level features from raw sensory data after using statistical learning over a large amount of data to obtain an effective representation of an input space. This is different from earlier approaches that use hand-crafted features or rules designed by experts.

The superior accuracy of DNNs, however, comes at the cost of high computational complexity. While general-purpose compute engines, especially graphics processing units (GPUs), have been the mainstay for much DNN processing, increasingly there is interest in providing more specialized acceleration of the DNN computation. This article aims to provide an overview of DNNs, the various tools for understanding their behavior, and the techniques being explored to efficiently accelerate their computation.

This paper is organized as follows:

- Section II provides background on the context of why DNNs are important, their history and applications.
- Section III gives an overview of the basic components of DNNs and popular DNN models currently in use.
- Section IV describes the various resources used for DNN research and development.
- Section V describes the various hardware platforms used to process DNNs and the various optimizations used to improve throughput and energy efficiency without impacting application accuracy (i.e., produce bit-wise identical results).
- Section VI discusses how mixed-signal circuits and new memory technologies can be used for near-data processing to address the expensive data movement that dominates throughput and energy consumption of DNNs.
- Section VII describes various joint algorithm and hardware optimizations that can be performed on DNNs to improve both throughput and energy efficiency while trying to minimize impact on accuracy.
- Section VIII describes the key metrics that should be considered when comparing various DNN designs.

II. Background on Deep Neural Networks (DNN)

In this section, we describe the position of DNNs in the context of AI in general and some of the concepts that motivated its development. We will also present a brief chronology of the major steps in its history, and some current domains to which it is being applied.

A. Artificial Intelligence and DNNs

DNNs, also referred to as deep learning, are a part of the broad field of AI, which is the science and engineering of creating intelligent machines that have the ability to

V. Sze, Y.-H. Chen and T.-J. Yang are with the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139 USA. (e-mail: sze@mit.edu; yhchen@mit.edu, tjy@mit.edu)

J. S. Emer is with the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139 USA, and also with Nvidia Corporation, Westford, MA 01886 USA. (e-mail: jsemer@mit.edu)

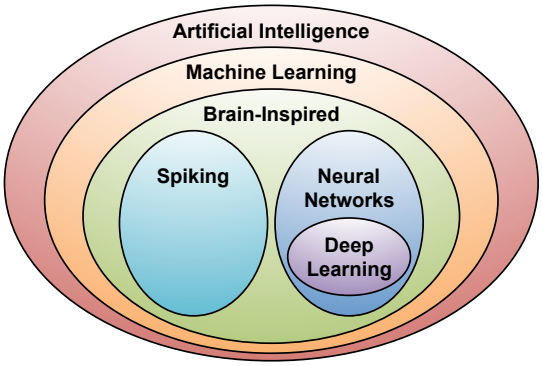


Fig. 1. Deep Learning in the context of Artificial Intelligence.

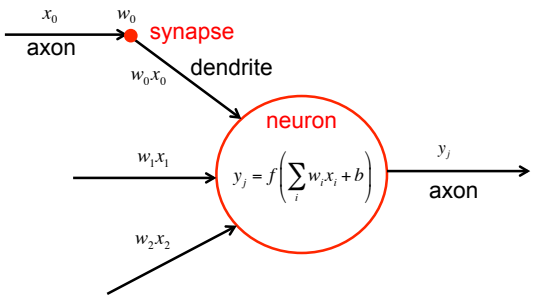


Fig. 2. Connections to a neuron in the brain. x_i, w_i, $f(\cdot)$, and b are the activations, weights, non-linear function and bias, respectively. (Figure adopted from [7].)

achieve goals like humans do, according to John McCarthy, the computer scientist who coined the term in the 1950s. The relationship of deep learning to the whole of artificial intelligence is illustrated in Fig. 1.

Within artificial intelligence is a large sub-field called machine learning, which was defined in 1959 by Arthur Samuel as the field of study that gives computers the ability to learn without being explicitly programmed. That means a single program, once created, will be able to learn how to do some intelligent activities outside the notion of programming. This is in contrast to purpose-built programs whose behavior is defined by hand-crafted heuristics that explicitly and statically define their behavior.

The advantage of an effective machine learning algorithm is clear. Instead of the laborious and hit-or-miss approach of creating a distinct, custom program to solve each individual problem in a domain, the single machine learning algorithm simply needs to learn, via a processes called *training*, to handle each new problem.

Within the machine learning field, there is an area that is often referred to as brain-inspired computation. Since the brain is currently the best 'machine' we know for learning and solving problems, it is a natural place to look for a machine learning approach. Therefore, a brain-inspired computation is a program or algorithm that takes some aspects of its basic form or functionality from the way the brain works. This is in contrast to attempts to create a brain, but rather the program aims to emulate some aspects of how we understand the brain to operate.

Although scientists are still exploring the details of how the brain works, it is generally believed that the main computational element of the brain is the *neuron*. There are approximately 86 billion neurons in the average human brain. The neurons themselves are connected together with a number of elements entering them called dendrites and an element leaving them called an axon as shown in Fig. 2. The neuron accepts the signals entering it via the dendrites, performs a computation on those signals, and generates a signal on the axon. These input and output signals are referred to as *activations*. The axon of one neuron branches out and is connected to the dendrites of many other neurons. The connections between a branch of the axon and a dendrite is called a *synapse*. There are estimated to be 10^{14} to 10^{15} synapses in the average human brain.

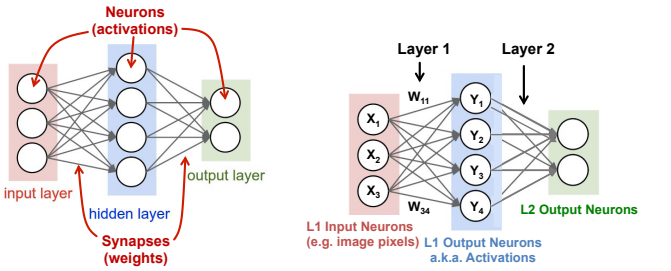
A key characteristic of the synapse is that it can scale the signal (x_i) crossing it as shown in Fig. 2. That scaling factor can be referred to as a *weight* (w_i), and the way the brain is believed to learn is through changes to the weights associated with the synapses. Thus, different weights result in different responses to an input. Note that learning is the adjustment of the weights in response to a learning stimulus, while the organization (what might be thought of as the program) of the brain does not change. This characteristic makes the brain an excellent inspiration for a machine-learning-style algorithm.

Within the brain-inspired computing paradigm there is a subarea called spiking computing. In this subarea, inspiration is taken from the fact that the communication on the dendrites and axons are spike-like pulses and that the information being conveyed is not just based on a spike's amplitude. Instead, it also depends on the time the pulse arrives and that the computation that happens in the neuron is a function of not just a single value but the width of pulse and the timing relationship between different pulses. An example of a project that was inspired by the spiking of the brain is the IBM TrueNorth [8]. In contrast to spiking computing, another subarea of brain-inspired computing is called neural networks, which is the focus of this article.[1]

B. Neural Networks and Deep Neural Networks (DNNs)

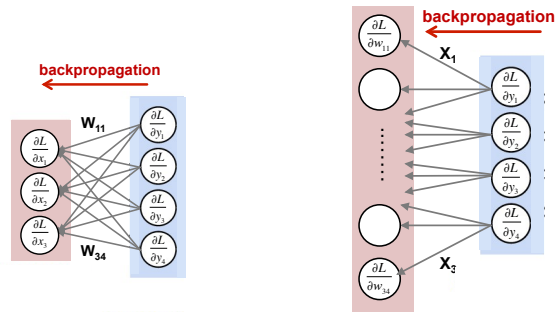
Neural networks take their inspiration from the notion that a neuron's computation involves a weighted sum of the input values. These weighted sums correspond to the value scaling performed by the synapses and the combining of those values in the neuron. Furthermore, the neuron doesn't just output that weighted sum, since the computation associated with a cascade of neurons would then be a simple linear algebra operation. Instead there is a functional operation within the neuron that is performed on the combined inputs. This operation appears to be a non-linear function that causes a neuron to generate an output only if the inputs cross some threshold. Thus by analogy, neural networks apply a non-linear function to the weighted sum of the input values. We look at what some of those non-linear functions are in Section III-A1.

[1]Note: Recent work using TrueNorth in a stylized fashion allows it to be used to compute reduced precision neural networks [9]. These types of neural networks are discussed in Section VII-A.



(a) Neurons and synapses (b) Compute weighted sum for each layer

Fig. 3. Simple neural network example and terminology (Figure adopted from [7]).



(a) Compute the gradient of the loss relative to the filter inputs (b) Compute the gradient of the loss relative to the weights

Fig. 4. An example of backpropagation through a neural network.

Fig. 3(a) shows a diagrammatic picture of a computational neural network. The neurons in the input layer receive some values and propagate them to the neurons in the middle layer of the network, which is also frequently called a 'hidden layer'. The weighted sums from one or more hidden layers are ultimately propagated to the output layer, which presents the final outputs of the network to the user. To align brain-inspired terminology with neural networks, the outputs of the neurons are often referred to as *activations*, and the synapses are often referred to as *weights* as shown in Fig. 3(a). We will use the activation/weight nomenclature in this article.

Fig. 3(b) shows an example of the computation at each layer: $y_j = f(\sum_{i=1}^{3} W_{ij} \times x_i + b)$, where W_{ij}, x_i and y_j are the weights, input activations and output activations, respectively, and $f(\cdot)$ is a non-linear function described in Section III-A1. The bias term b is omitted from Fig. 3(b) for simplicity.

Within the domain of neural networks, there is an area called *deep learning*, in which the neural networks have more than three layers, i.e., more than one hidden layer. Today, the typical numbers of network layers used in deep learning range from five to more than a thousand. In this article, we will generally use the terminology *deep neural networks (DNNs)* to refer to the neural networks used in deep learning.

DNNs are capable of learning high-level features with more complexity and abstraction than shallower neural networks. An example that demonstrates this point is using DNNs to process visual data. In these applications, pixels of an image are fed into the first layer of a DNN, and the outputs of that layer can be interpreted as representing the presence of different low-level features in the image, such as lines and edges. At subsequent layers, these features are then combined into a measure of the likely presence of higher level features, e.g., lines are combined into shapes, which are further combined into sets of shapes. And finally, given all this information, the network provides a probability that these high-level features comprise a particular object or scene. This deep feature hierarchy enables DNNs to achieve superior performance in many tasks.

C. Inference versus Training

Since DNNs are an instance of a machine learning algorithm, the basic program does not change as it learns to perform its given tasks. In the specific case of DNNs, this learning involves determining the value of the weights (and bias) in the network, and is referred to as *training* the network. Once trained, the program can perform its task by computing the output of the network using the weights determined during the training process. Running the program with these weights is referred to as *inference*.

In this section, we will use image classification, as shown in Fig. 6, as a driving example for training and using a DNN. When we perform inference using a DNN, we give an input image and the output of the DNN is a vector of scores, one for each object class; the class with the highest score indicates the most likely class of object in the image. The overarching goal for training a DNN is to determine the weights that maximize the score of the correct class and minimize the scores of the incorrect classes. When training the network the correct class is often known because it is given for the images used for training (i.e., the training set of the network). The gap between the ideal correct scores and the scores computed by the DNN based on its current weights is referred to as the *loss* (L). Thus the goal of training DNNs is to find a set of weights to minimize the average loss over a large training set.

When training a network, the weights (w_{ij}) are usually updated using a hill-climbing optimization process called gradient descent. A multiple of the gradient of the loss relative to each weight, which is the partial derivative of the loss with respect to the weight, is used to update the weight (i.e., updated $w_{ij}^{t+1} = w_{ij}^{t} - \alpha \frac{\partial L}{\partial w_{ij}}$, where α is called the learning rate). Note that this gradient indicates how the weights should change in order to reduce the loss. The process is repeated iteratively to reduce the overall loss.

An efficient way to compute the partial derivatives of the gradient is through a process called *backpropagation*. Backpropagation, which is a computation derived from the *chain rule* of calculus, operates by passing values backwards through the network to compute how the loss is affected by each weight.

This backpropagation computation is, in fact, very similar in form to the computation used for inference as shown in Fig. 4 [10].[2] Thus, techniques for efficiently performing

[2]To backpropagate through each filter: (1) compute the gradient of the loss relative to the weights from the filter inputs (i.e., the forward activations) and the gradients of the loss relative to the filter outputs; (2) compute the gradient of the loss relative to the filter inputs from the filter weights and the gradients of the loss relative to the filter outputs.

inference can sometimes be useful for performing training. It is, however, important to note a couple of points. First, backpropagation requires intermediate outputs of the network to be preserved for the backwards computation, thus training has increased storage requirements. Second, due to the gradients use for hill-climbing, the precision requirement for training is generally higher than inference. Thus many of the reduced precision techniques discussed in Section VII are limited to inference only.

A variety of techniques are used to improve the efficiency and robustness of training. For example, often the loss from multiple sets of input data, i.e., a *batch*, are collected before a single pass of weight update is performed; this helps to speed up and stabilize the training process.

There are multiple ways to train the weights. The most common approach, as described above, is called *supervised learning*, where all the training samples are labeled (e.g., with the correct class). *Unsupervised learning* is another approach where all the training samples are not labeled and essentially the goal is to find the structure or clusters in the data. *Semi-supervised learning* falls in between the two approaches where only a small subset of the training data is labeled (e.g., use unlabeled data to define the cluster boundaries, and use the small amount of labeled data to label the clusters). Finally, *reinforcement learning* can be used to the train weights such that given the state of the current environment, the DNN can output what action the agent should take next to maximize expected rewards; however, the rewards might not be available immediately after an action, but instead only after a series of actions.

Another commonly used approach to determine weights is *fine-tuning*, where previously-trained weights are available and are used as a starting point and then those weights are adjusted for a new dataset (e.g., transfer learning) or for a new constraint (e.g., reduced precision). This results in faster training than starting from a random starting point, and can sometimes result in better accuracy.

This article will focus on the efficient processing of DNN inference rather than training, since DNN inference is often performed on embedded devices (rather than the cloud) where resources are limited as discussed in more details later.

D. Development History

Although neural nets were proposed in the 1940s, the first practical application employing multiple digital neurons didn't appear until the late 1980s with the LeNet network for hand-written digit recognition [11][3]. Such systems are widely used by ATMs for digit recognition on checks. However, the early 2010s have seen a blossoming of DNN-based applications with highlights such as Microsoft's speech recognition system in 2011 [2] and the AlexNet system for image recognition in 2012 [3]. A brief chronology of deep learning is shown in Fig. 5.

The deep learning successes of the early 2010s are believed to be a confluence of three factors. The first factor is the amount of available information to train the networks. To learn a powerful representation (rather than using a hand-crafted approach) requires a large amount of training data. For example, Facebook receives over 350 millions images per day, Walmart creates 2.5 Petabytes of customer data hourly and YouTube has 300 hours of video uploaded every minute. As a result, the cloud providers and many businesses have a huge amount of data to train their algorithms.

DNN Timeline

- 1940s - Neural networks were proposed
- 1960s - Deep neural networks were proposed
- 1989 - Neural networks for recognizing digits (LeNet)
- 1990s - Hardware for shallow neural nets (Intel ETANN)
- 2011 - Breakthrough DNN-based speech recognition (Microsoft)
- 2012 - DNNs for vision start supplanting hand-crafted approaches (AlexNet)
- 2014+ - Rise of DNN accelerator research (Neuflow, DianNao...)

Fig. 5. A concise history of neural networks. 'Deep' refers to the number of layers in the network.

The second factor is the amount of compute capacity available. Semiconductor device and computer architecture advances have continued to provide increased computing capability, and we appear to have crossed a threshold where the large amount of weighted sum computation in DNNs, which is required for both inference and training, can be performed in a reasonable amount of time.

The successes of these early DNN applications opened the floodgates of algorithmic development. It has also inspired the development of several (largely open source) frameworks that make it even easier for researchers and practitioners to explore and use DNNs. Combining these efforts contributes to the third factor, which is the evolution of the algorithmic techniques that have improved application accuracy significantly and broadened the domains to which DNNs are being applied.

An excellent example of the successes in deep learning can be illustrated with the ImageNet Challenge [14]. This challenge is a contest involving several different components. One of the components is an image classification task where algorithms are given an image and they must identify what is in the image, as shown in Fig. 6. The training set consists of 1.2 million images, each of which is labeled with one of 1000 object categories that the image contains. For the evaluation phase, the algorithm must accurately identify objects in a test set of images, which it hasn't previously seen.

Fig. 7 shows the performance of the best entrants in the ImageNet contest over a number of years. One sees that the accuracy of the algorithms initially had an error rate of 25% or more. In 2012, a group from the University of Toronto used graphics processing units (GPUs) for their high compute capability and a deep neural network approach, named AlexNet, and dropped the error rate by approximately 10% [3]. Their accomplishment inspired an outpouring of deep learning style algorithms that have resulted in a steady stream of

[3]In the early 1960s, single analog neuron systems were used for adaptive filtering [12, 13].

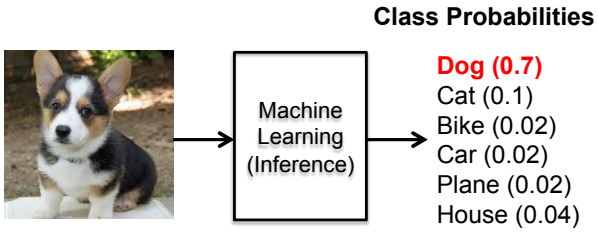


Fig. 6. Example of an image classification task. The machine learning platform takes in an image and outputs the confidence scores for a predefined set of classes.

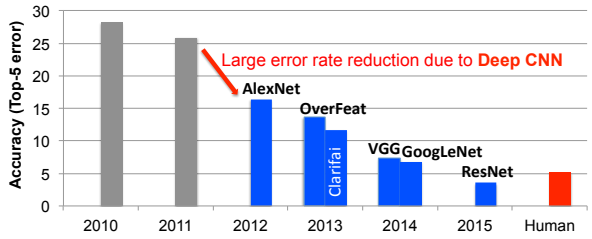


Fig. 7. Results from the ImageNet Challenge [14].

improvements.

In conjunction with the trend to deep learning approaches for the ImageNet Challenge, there has been a corresponding increase in the number of entrants using GPUs. From 2012 when only 4 entrants used GPUs to 2014 when almost all the entrants (110) were using them. This reflects the almost complete switch from traditional computer vision approaches to deep learning-based approaches for the competition.

In 2015, the ImageNet winning entry, ResNet [15], exceeded human-level accuracy with a top-5 error rate[4] below 5%. Since then, the error rate has dropped below 3% and more focus is now being placed on more challenging components of the competition, such as object detection and localization. These successes are clearly a contributing factor to the wide range of applications to which DNNs are being applied.

E. Applications of DNN

Many applications can benefit from DNNs ranging from multimedia to medical space. In this section, we will provide examples of areas where DNNs are currently making an impact and highlight emerging areas where DNNs hope to make an impact in the future.

- **Image and Video** Video is arguably the biggest of the big data. It accounts for over 70% of today's Internet traffic [16]. For instance, over 800 million hours of video is collected daily worldwide for video surveillance [17]. Computer vision is necessary to extract meaningful information from video. DNNs have significantly improved the accuracy of many computer vision tasks such as image classification [14], object localization and detection [18], image segmentation [19], and action recognition [20].
- **Speech and Language** DNNs have significantly improved the accuracy of speech recognition [21] as well as many related tasks such as machine translation [2], natural language processing [22], and audio generation [23].
- **Medical** DNNs have played an important role in genomics to gain insight into the genetics of diseases such as autism, cancers, and spinal muscular atrophy [24–27]. They have also been used in medical imaging to detect skin cancer [5], brain cancer [28] and breast cancer [29].
- **Game Play** Recently, many of the grand AI challenges involving game play have been overcome using DNNs. These successes also required innovations in training techniques and many rely on reinforcement learning [30]. DNNs have surpassed human level accuracy in playing Atari [31] as well as Go [6], where an exhaustive search of all possibilities is not feasible due to the unimaginably huge number of possible moves.
- **Robotics** DNNs have been successful in the domain of robotic tasks such as grasping with a robotic arm [32], motion planning for ground robots [33], visual navigation [4, 34], control to stabilize a quadcopter [35] and driving strategies for autonomous vehicles [36].

DNNs are already widely used in multimedia applications today (e.g., computer vision, speech recognition). Looking forward, we expect that DNNs will likely play an increasingly important role in the medical and robotics fields, as discussed above, as well as finance (e.g., for trading, energy forecasting, and risk assessment), infrastructure (e.g., structural safety, and traffic control), weather forecasting and event detection [37]. The myriad application domains pose new challenges to the efficient processing of DNNs; the solutions then have to be adaptive and scalable in order to handle the new and varied forms of DNNs that these applications may employ.

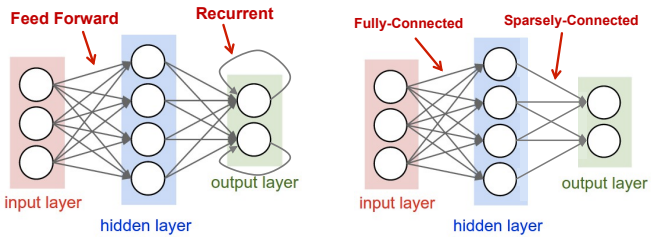
F. Embedded versus Cloud

The various applications and aspects of DNN processing (i.e., training versus inference) have different computational needs. Specifically, training often requires a large dataset[5] and significant computational resources for multiple weight-update iterations. In many cases, training a DNN model still takes several hours to multiple days and thus is typically performed in the cloud. Inference, on the other hand, can happen either in the cloud or at the edge (e.g., IoT or mobile).

In many applications, it is desirable to have the DNN inference processing near the sensor. For instance, in computer vision applications, such as measuring wait times in stores or predicting traffic patterns, it would be desirable to extract meaningful information from the video right at the image sensor rather than in the cloud to reduce the communication cost. For other applications such as autonomous vehicles, drone navigation and robotics, local processing is desired since the latency and security risks of relying on the cloud are too high. However, video involves a large amount of data, which is computationally complex to process; thus, low cost hardware to analyze video is challenging yet critical to enabling

[4]The top-5 error rate is measured based on whether the correct answer appears in one of the top 5 categories selected by the algorithm.

[5]One of the major drawbacks of DNNs is their need for large datasets to prevent over-fitting during training.



(a) Feedforward versus feedback (recurrent) networks

(b) Fully connected versus sparse

Fig. 8. Different types of neural networks (Figure adopted from [7]).

these applications. Speech recognition enables us to seamlessly interact with electronic devices, such as smartphones. While currently most of the processing for applications such as Apple Siri and Amazon Alexa voice services is in the cloud, it is still desirable to perform the recognition on the device itself to reduce latency and dependency on connectivity, and to improve privacy and security.

Many of the embedded platforms that perform DNN inference have stringent energy consumption, compute and memory cost limitations; efficient processing of DNNs have thus become of prime importance under these constraints. Therefore, in this article, we will focus on the compute requirements for inference rather than training.

III. Overview of DNNs

DNNs come in a wide variety of shapes and sizes depending on the application. The popular shapes and sizes are also evolving rapidly to improve accuracy and efficiency. In all cases, the input to a DNN is a set of values representing the information to be analyzed by the network. For instance, these values can be pixels of an image, sampled amplitudes of an audio wave or the numerical representation of the state of some system or game.

The networks that process the input come in two major forms: feed forward and recurrent as shown in Fig. 8(a). In feed-forward networks all of the computation is performed as a sequence of operations on the outputs of a previous layer. The final set of operations generates the output of the network, for example a probability that an image contains a particular object, the probability that an audio sequence contains a particular word, a bounding box in an image around an object or the proposed action that should be taken. In such DNNs, the network has no memory and the output for an input is always the same irrespective of the sequence of inputs previously given to the network.

In contrast, recurrent neural networks (RNNs), of which Long Short-Term Memory networks (LSTMs) [38] are a popular variant, have internal memory to allow long-term dependencies to affect the output. In these networks, some intermediate operations generate values that are stored internally in the network and used as inputs to other operations in conjunction with the processing of a later input. In this article, we will focus on feed-forward networks since (1) the major computation in RNNs is still the weighted sum, which is covered by the feed-forward networks, and (2) to-date little attention has been given to hardware acceleration specifically for RNNs.

DNNs can be composed solely of *fully-connected* (FC) layers (also referred to as multi-layer perceptrons, or MLP) as shown in the leftmost layer of Fig. 8(b). In a FC layer, all output activations are composed of a weighted sum of all input activations (i.e., all outputs are connected to all inputs). This requires a significant amount of storage and computation. Thankfully, in many applications, we can remove some connections between the activations by setting the weights to zero without affecting accuracy. This results in a *sparsely-connected layer*. A sparsely connected layer is illustrated in the rightmost layer of Fig. 8(b).

We can also make the computation more efficient by limiting the number of weights that contribute to an output. This sort of structured sparsity can arise if each output is only a function of a fixed-size window of inputs. Even further efficiency can be gained if the same set of weights are used in the calculation of every output. This repeated use of the same weight values is called *weight sharing* and can significantly reduce the storage requirements for weights.

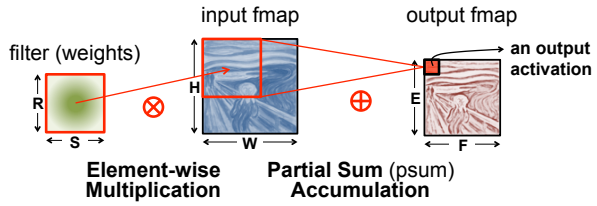
An extremely popular windowed and weight-shared DNN layer arises by structuring the computation as a convolution, as shown in Fig. 9(a), where the weighted sum for each output activation is computed using only a small neighborhood of input activations (i.e., all weights beyond beyond the neighborhood are set to zero), and where the same set of weights are shared for every output (i.e., the filter is space invariant). Such convolution-based layers are referred to as *convolutional* (CONV) layers. [6]

A. Convolutional Neural Networks (CNNs)

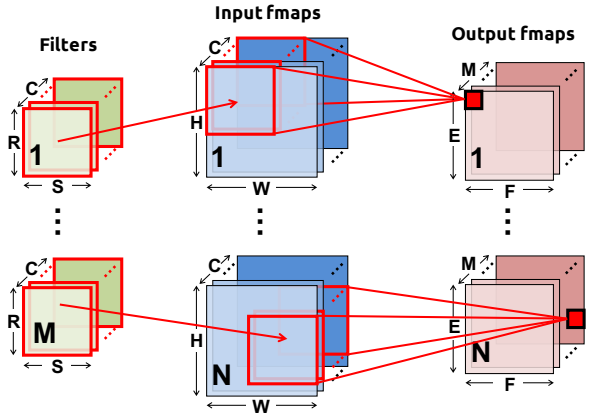
A common form of DNNs is *Convolutional Neural Nets* (CNNs), which are composed of multiple CONV layers as shown in Fig. 10. In such networks, each layer generates a successively higher-level abstraction of the input data, called a *feature map* (fmap), which preserves essential yet unique information. Modern CNNs are able to achieve superior performance by employing a very deep hierarchy of layers. CNN are widely used in a variety of applications including image understanding [3], speech recognition [39], game play [6], robotics [32], etc. This paper will focus on its use in image processing, specifically for the task of image classification [3].

Each of the CONV layers in CNN is primarily composed of high-dimensional convolutions as shown in Fig. 9(b). In this computation, the input activations of a layer are structured as a set of 2-D *input feature maps* (ifmaps), each of which is called a *channel*. Each channel is convolved with a distinct 2-D filter from the stack of filters, one for each channel; this stack of 2-D filters is often referred to as a single 3-D filter. The results of the convolution at each point are summed across all the channels. In addition, a 1-D bias can be added to the filtering results, but some recent networks [15] remove its usage from parts of the layers. The result of this computation is the output activations that comprise one channel of *output feature map* (ofmap). Additional 3-D filters can be used on

[6]Note: the structured sparsity in CONV layers is orthogonal to the sparsity that occurs from network pruning as described in Section VII-B2.



(a) 2-D convolution in traditional image processing



(b) High dimensional convolutions in CNNs

Fig. 9. Dimensionality of convolutions.

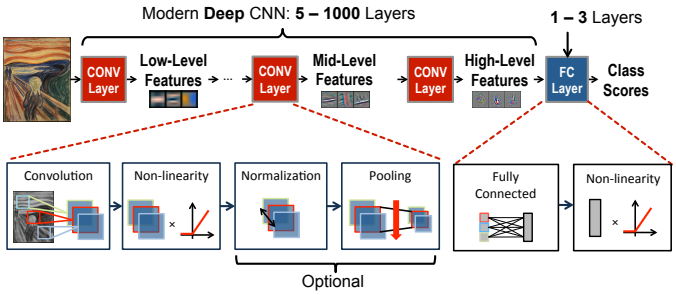


Fig. 10. Convolutional Neural Networks.

the same input to create additional output channels. Finally, multiple input feature maps may be processed together as a *batch* to potentially improve reuse of the filter weights.

Given the shape parameters in Table I, the computation of a CONV layer is defined as

$$\mathbf{O}[z][u][x][y] = \mathbf{B}[u] + \sum_{k=0}^{C-1}\sum_{i=0}^{S-1}\sum_{j=0}^{R-1} \mathbf{I}[z][k][Ux+i][Uy+j] \times \mathbf{W}[u][k][i][j],$$
$$0 \le z < N, 0 \le u < M, 0 \le x < F, 0 \le y < E,$$
$$E = (H - R + U)/U, F = (W - S + U)/U. \quad (1)$$

$\mathbf{O}$, $\mathbf{I}$, $\mathbf{W}$ and $\mathbf{B}$ are the matrices of the ofmaps, ifmaps, filters and biases, respectively. U is a given stride size. Fig. 9(b) shows a visualization of this computation (ignoring biases).

To align the terminology of CNNs with the generic DNN,

- filters are composed of weights (i.e., synapses)
- input and output feature maps (ifmaps, ofmaps) are composed of activations (i.e., input and output neurons)

Shape Parameter	Description
N	batch size of 3-D fmaps
M	# of 3-D filters / # of ofmap channels
C	# of ifmap/filter channels
H/W	ifmap plane height/width
R/S	filter plane height/width (= H or W in FC)
E/F	ofmap plane height/width (= 1 in FC)

TABLE I
SHAPE PARAMETERS OF A CONV/FC LAYER.

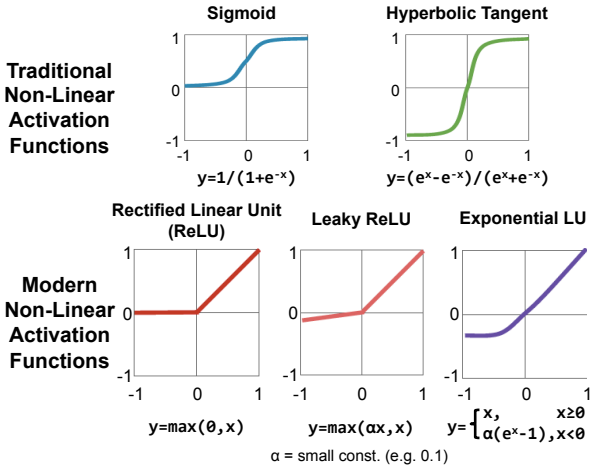


Fig. 11. Various forms of non-linear activation functions (Figure adopted from Caffe Tutorial [46]).

From five [3] to more than a thousand [15] CONV layers are commonly used in recent CNN models. A small number, e.g., 1 to 3, of fully-connected (FC) layers are typically applied after the CONV layers for classification purposes. A FC layer also applies filters on the ifmaps as in the CONV layers, but the filters are of the same size as the ifmaps. Therefore, it does not have the weight sharing property of CONV layers. Eq. (1) still holds for the computation of FC layers with a few additional constraints on the shape parameters: $H = R$, $F = S$, $E = F = 1$, and $U = 1$.

In addition to CONV and FC layers, various optional layers can be found in a DNN such as the non-linearity, pooling, and normalization. The function and computations for each of these layers are discussed next.

1) Non-Linearity: A non-linear activation function is typically applied after each CONV or FC layer. Various non-linear functions are used to introduce non-linearity into the DNN as shown in Fig. 11. These include historically conventional non-linear functions such as sigmoid or hyperbolic tangent as well as rectified linear unit (ReLU) [40], which has become popular in recent years due to its simplicity and its ability to enable fast training. Variations of ReLU, such as leaky ReLU [41], parametric ReLU [42], and exponential LU [43] have also been explored for improved accuracy. Finally, a non-linearity called maxout, which takes the max value of two intersecting linear functions, has shown to be effective in speech recognition tasks [44, 45].

2) Pooling: A variety of computations that reduce the dimensionality of a feature map are referred to as *pooling*. Pooling, which is applied to each channel separately, enables

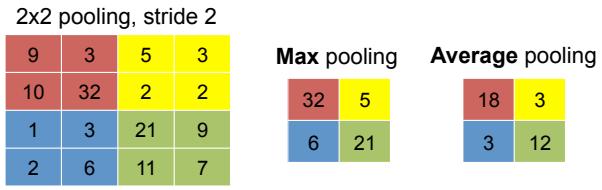


Fig. 12. Various forms of pooling (Figure adopted from Caffe Tutorial [46]).

the network to be robust and invariant to small shifts and distortions. Pooling combines, or *pools*, a set of values in its *receptive field* into a smaller number of values. It can be configured based on the size of its receptive field (e.g., 2×2) and pooling operation (e.g., max or average), as shown in Fig. 12. Typically pooling occurs on non-overlapping blocks (i.e., the stride is equal to the size of the pooling). Usually a stride of greater than one is used such that there is a reduction in the dimension of the representation (i.e., feature map).

3) Normalization: Controlling the input distribution across layers can help to significantly speed up training and improve accuracy. Accordingly, the distribution of the layer input activations (σ, μ) are normalized such that it has a zero mean and a unit standard deviation. In batch normalization (BN), the normalized value is further scaled and shifted, as shown in Eq. (2), where the parameters (γ, β) are learned from training [47]. ϵ is a small constant to avoid numerical problems. Prior to this, local response normalization (LRN) [3] was used, which was inspired by lateral inhibition in neurobiology where excited neurons (i.e., high value activations) should subdue its neighbors (i.e., cause low value activations); however, BN is now considered standard practice in the design of CNNs while LRN is mostly deprecated. Note that while LRN usually is performed after the non-linear function, BN is mostly performed between the CONV or FC layer and the non-linear function.

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta \tag{2}$$

B. Popular DNN Models

Many DNN models have been developed over the past two decades. Each of these models has a different 'network architecture' in terms of number of layers, layer types, layer shapes (i.e., filter size, number of channels and filters), and connections between layers. Understanding these variations and trends is important for incorporating the right flexibility in any efficient DNN engine.

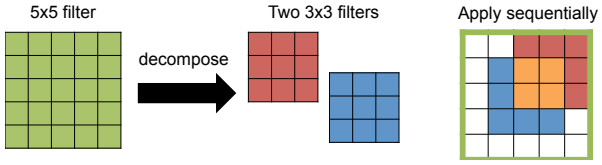
In this section, we will give an overview of various popular DNNs such as LeNet [48] as well as those that competed in and/or won the ImageNet Challenge [14] as shown in Fig. 7, most of whose models with pre-trained weights are publicly available for download; the DNN models are summarized in Table II. Two results for top-5 error results are reported. In the first row, the accuracy is boosted by using multiple crops from the image and an ensemble of multiple trained models (i.e., the DNN needs to be run several times); these results were used to compete in the ImageNet Challenge. The second row reports the accuracy if only a single crop was used (i.e., the DNN is run only once), which is more consistent with what would likely be deployed in real-time and/or energy-constrained applications.

LeNet [11] was one of the first CNN approaches introduced in 1989. It was designed for the task of digit classification in grayscale images of size 28×28. The most well known version, LeNet-5, contains two CONV layers and two FC layers [48]. Each CONV layer uses filters of size 5×5 (1 channel per filter) with 6 filters in the first layer and 16 filters in the second layer. Average pooling of 2×2 is used after each convolution and a sigmoid is used for the non-linearity. In total, LeNet requires 60k weights and 341k multiply-and-accumulates (MACs) per image. LeNet led to CNNs' first commercial success, as it was deployed in ATMs to recognize digits for check deposits.

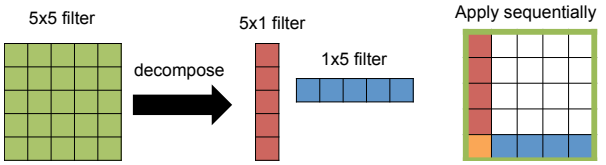
AlexNet [3] was the first CNN to win the ImageNet Challenge in 2012. It consists of five CONV layers and three FC layers. Within each CONV layer, there are 96 to 384 filters and the filter size ranges from 3×3 to 11×11, with 3 to 256 channels each. In the first layer, the 3 channels of the filter correspond to the red, green and blue components of the input image. A ReLU non-linearity is used in each layer. Max pooling of 3×3 is applied to the outputs of layers 1, 2 and 5. To reduce computation, a stride of 4 is used at the first layer of the network. AlexNet introduced the use of LRN in layers 1 and 2 before the max pooling, though LRN is no longer popular in later CNN models. One important factor that differentiates AlexNet from LeNet is that the number of weights is much larger and the shapes vary from layer to layer. To reduce the amount of weights and computation in the second CONV layer, the 96 output channels of the first layer are split into two groups of 48 input channels for the second layer, such that the filters in the second layer only have 48 channels. Similarly, the weights in fourth and fifth layer are also split into two groups. In total, AlexNet requires 61M weights and 724M MACs to process one 227×227 input image.

Overfeat [49] has a very similar architecture to AlexNet with five CONV layers and three FC layers. The main differences are that the number of filters is increased for layers 3 (384 to 512), 4 (384 to 1024), and 5 (256 to 1024), layer 2 is not split into two groups, the first fully connected layer only has 3072 channels rather than 4096, and the input size is 231×231 rather than 227×227. As a result, the number of weights grows to 146M and the number of MACs grows to 2.8G per image. Overfeat has two different models: fast (described here) and accurate. The accurate model used in the ImageNet Challenge gives a 0.65% lower top-5 error rate than the fast model at the cost of 1.9× more MACs

VGG-16 [50] goes deeper to 16 layers consisting of 13 CONV layers and 3 FC layers. In order to balance out the cost of going deeper, larger filters (e.g., 5×5) are built from multiple smaller filters (e.g., 3×3), which have fewer weights, to achieve the same receptive fields as shown in Fig. 13(a). As a result, all CONV layers have the same filter size of 3×3. In total, VGG-16 requires 138M weights and 15.5G MACs to process one 224×224 input image. VGG has two different models: VGG-16 (described here) and VGG-19. VGG-19 gives a 0.1% lower top-5 error rate than VGG-16 at the cost of 1.27× more MACs.



(a) Constructing a 5×5 support from 3×3 filters. Used in VGG-16.



(b) Constructing a 5×5 support from 1×5 and 5×1 filter. Used in GoogleNet/Inception v3 and v4.

Fig. 13. Decomposing larger filters into smaller filters.

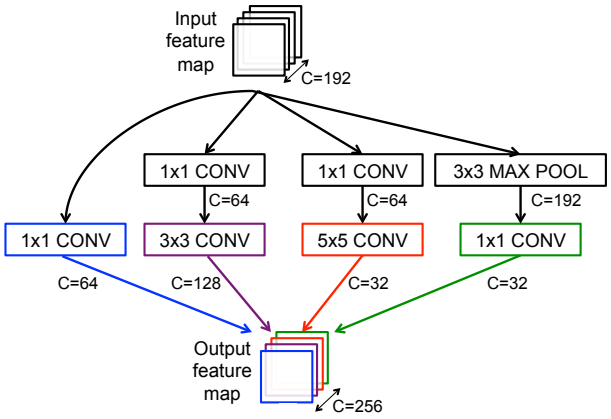
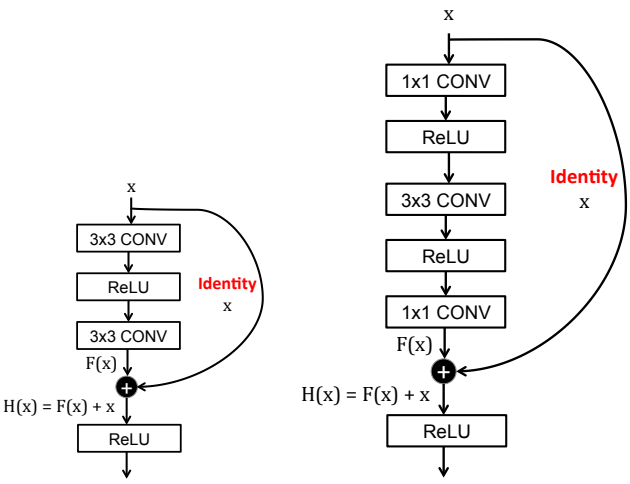


Fig. 14. Inception module from GoogleNet [51] with example channel lengths. Note that each CONV layer is followed by a ReLU (not drawn).



(a) Without bottleneck (b) With bottleneck

Fig. 15. Shortcut module from ResNet [15]. Note that ReLU following last CONV layer in short cut is *after* the addition.

GoogLeNet [51] goes even deeper with 22 layers. It introduced an inception module, shown in Fig. 14, which is composed of parallel connections, whereas previously there was only a single serial connection. Different sized filters (i.e., 1×1, 3×3, 5×5), along with 3×3 max-pooling, are used for each parallel connection and their outputs are concatenated for the module output. Using multiple filter sizes has the effect of processing the input at multiple scales. For improved training speed, GoogLeNet is designed such that the weights and the activations, which are stored for backpropagation during training, could all fit into the GPU memory. In order to reduce the number of weights, 1×1 filters are applied as a 'bottleneck' to reduce the number of channels for each filter [52]. The 22 layers consist of three CONV layers, followed by 9 inceptions layers (each of which are two CONV layers deep), and one FC layer. Since its introduction in 2014, GoogleNet (also referred to as Inception) has multiple versions: v1 (described here), v3 [7] and v4. Inception-v3 decomposes the convolutions by using smaller 1-D filters as shown in Fig. 13(b) to reduce number of MACs and weights in order to go deeper to 42 layers. In conjunction with batch normalization [47], v3 achieves over 3% lower top-5 error than v1 with 2.5× increase in computation [53]. Inception-v4 uses residual connections [54], described in the next section, for a 0.4% reduction in error.

ResNet [15], also known as Residual Net, uses residual connections to go even deeper (34 layers or more). It was the first entry DNN in ImageNet Challenge that exceeded human-level accuracy with a top-5 error rate below 5%. One of the challenges with deep networks is the vanishing gradient during training: as the error backpropagates through the network the gradient shrinks, which affects the ability to update the weights in the earlier layers for very deep networks. Residual net introduces a 'shortcut' module which contains an identity connection such that the weight layers (i.e., CONV layers) can be skipped as shown in Fig. 15. Rather than learning the function for the weight layers $F(x)$, the shortcut module learns the residual mapping ($F(x) = H(x) - x$). Initially, $F(x)$ is zero and the identity connection is taken; then gradually during training, the actual forward connection through the weight layer is used. This is similar to the LSTM networks that are used for sequential data. ResNet also uses the 'bottleneck' approach of using 1×1 filters to reduce the number of weight parameters. As a result, the two layers in the shortcut module are replaced by three layers (1×1, 3×3, 1×1) where the 1×1 reduces and then increases (restores) the number of weights. ResNet-50 consists of one CONV layer, followed by 16 shortcut layers (each of which are three CONV layers deep), and one FC layer; it requires 25.5M weights and 3.9G MACs per image. There are various versions of ResNet with multiple depths (e.g., *without bottleneck:* 18, 34; *with bottleneck:* 50, 101, 152). The ResNet with 152 layers was the winner of the ImageNet Challenge requiring 11.3G MACs and 60M weights. Compared to ResNet-50, it reduces the top-5 error by around 1% at the cost of 2.9× more MACs and 2.5× more weights.

Several trends can be observed in the popular DNNs shown in Table II. Increasing the depth of the network tends to provide higher accuracy. Controlling for number of weights, a deeper network can support a wider range of non-linear functions

[7] v2 is very similar to v3.

that are more discriminative and also provides more levels of hierarchy in the learned representation [15, 50, 51, 55]. The number of filter shapes continues to vary across layers, thus flexibility is still important. Furthermore, most of the computation has been placed on CONV layers rather than FC layers. In addition, the number of weights in the FC layers is reduced and in most recent networks (since GoogLeNet) the CONV layers also dominate in terms of weights. Thus, the focus of hardware implementations should be on addressing the efficiency of the CONV layers, which in many domains are increasingly important.

IV. DNN DEVELOPMENT RESOURCES

One of the key factors that has enabled the rapid development of DNNs is the set of development resources that have been made available by the research community and industry. These resources are also key to the development of DNN accelerators by providing characterizations of the workloads and facilitating the exploration of trade-offs in model complexity and accuracy. This section will describe these resources such that those who are interested in this field can quickly get started.

A. Frameworks

For ease of DNN development and to enable sharing of trained networks, several deep learning frameworks have been developed from various sources. These open source libraries contain software libraries for DNNs. Caffe was made available in 2014 from UC Berkeley [46]. It supports C, C++, Python and MATLAB. Tensorflow was released by Google in 2015, and supports C++ and python; it also supports multiple CPUs and GPUs and has more flexibility than Caffe, with the computation expressed as dataflow graphs to manage the tensors (multidimensional arrays). Another popular framework is Torch, which was developed by Facebook and NYU and supports C, C++ and Lua. There are several other frameworks such as Theano, MXNet, CNTK, which are described in [60]. There are also higher-level libraries that can run on top of the aforementioned frameworks to provide a more universal experience and faster development. One example of such libraries is Keras, which is written in Python and supports Tensorflow, CNTK and Theano.

The existence of such frameworks are not only a convenient aid for DNN researchers and application designers, but they are also invaluable for engineering high performance or more efficient DNN computation engines. In particular, because the frameworks make heavy use of a set primitive operations, such processing of a CONV layer, they can incorporate use of optimized software or hardware accelerators. This acceleration is transparent to the user of the framework. Thus, for example, most frameworks can use Nvidia's cuDNN library for rapid execution on Nvidia GPUs. Similarly, transparent incorporation of dedicated hardware accelerators can be achieved as was done with the Eyeriss chip [61].

Finally, these frameworks are a valuable source of workloads for hardware researchers. They can be used to drive experimental designs for different workloads, for profiling different workloads and for exploring hardware-software trade-offs.

B. Models

Pretrained DNN models can be downloaded from various websites [56–59] for the various different frameworks. It should be noted that even for the same DNN (e.g., AlexNet) the accuracy of these models can vary by around 1% to 2% depending on how the model was trained, and thus the results do not always exactly match the original publication.

C. Popular Datasets for Classification

It is important to factor in the difficulty of the task when comparing different DNN models. For instance, the task of classifying handwritten digits from the MNIST dataset [62] is much simpler than classifying an object into one of 1000 classes as is required for the ImageNet dataset [14](Fig. 16). It is expected that the size of the DNNs (i.e., number of weights) and the number of MACs will be larger for the more difficult task than the simpler task and thus require more energy and have lower throughput. For instance, LeNet-5[48] is designed for digit classification, while AlexNet[3], VGG-16[50], GoogLeNet[51], and ResNet[15] are designed for the 1000-class image classification.

There are many AI tasks that come with publicly available datasets in order to evaluate the accuracy of a given DNN. Public datasets are important for comparing the accuracy of different approaches. The simplest and most common task is image classification, which involves being given an entire image, and selecting 1 of N classes that the image most likely belongs to. There is no localization or detection.

MNIST is a widely used dataset for digit classification that was introduced in 1998 [62]. It consists of 28×28 pixel grayscale images of handwritten digits. There are 10 classes (for 10 digits) and 60,000 training images and 10,000 test images. LeNet-5 was able to achieve an accuracy of 99.05% when MNIST was first introduced. Since then the accuracy has increased to 99.79% using regularization of neural networks with dropconnect [63]. Thus, MNIST is now considered a fairly easy dataset.

CIFAR is a dataset that consists of 32×32 pixel colored images of of various objects, which was released in 2009 [64]. CIFAR is a subset of the 80 million Tiny Image dataset [65]. CIFAR-10 is composed of 10 mutually exclusive classes. There are 50,000 training images (5000 per class) and 10,000 test images (1000 per class). A two-layer convolutional deep belief network was able to achieve 64.84% accuracy on CIFAR-10 when it was first introduced [66]. Since then the accuracy has increased to 96.53% using fractional max pooling [67].

ImageNet is a large scale image dataset that was first introduced in 2010; the dataset stabilized in 2012 [14]. It contains images of 256×256 pixel in color with 1000 classes. The classes are defined using the WordNet as a backbone to handle ambiguous word meanings and to combine together synonyms into the same object category. In otherwords, there is a hierarchy for the ImageNet categories. The 1000 classes were selected such that there is no overlap in the ImageNet hierarchy. The ImageNet dataset contains many fine-grained categories including 120 different breeds of dogs. There are 1.3M training images (732 to 1300 per class), 100,000 testing

Metrics	LeNet 5	AlexNet	Overfeat fast	VGG 16	GoogLeNet v1	ResNet 50
Top-5 error†	n/a	16.4	14.2	7.4	6.7	5.3
Top-5 error (single crop)†	n/a	19.8	17.0	8.8	10.7	7.0
Input Size	28×28	227×227	231×231	224×224	224×224	224×224
# of CONV Layers	2	5	5	13	57	53
Depth in # of CONV Layers	2	5	5	13	21	49
Filter Sizes	5	3,5,11	3,5,11	3	1,3,5,7	1,3,7
# of Channels	1, 20	3-256	3-1024	3-512	3-832	3-2048
# of Filters	20, 50	96-384	96-1024	64-512	16-384	64-2048
Stride	1	1,4	1,4	1	1,2	1,2
Weights	2.6k	2.3M	16M	14.7M	6.0M	23.5M
MACs	283k	666M	2.67G	15.3G	1.43G	3.86G
# of FC Layers	2	3	3	3	1	1
Filter Sizes	1,4	1,6	1,6,12	1,7	1	1
# of Channels	50, 500	256-4096	1024-4096	512-4096	1024	2048
# of Filters	10, 500	1000-4096	1000-4096	1000-4096	1000	1000
Weights	58k	58.6M	130M	124M	1M	2M
MACs	58k	58.6M	130M	124M	1M	2M
Total Weights	60k	61M	146M	138M	7M	25.5M
Total MACs	341k	724M	2.8G	15.5G	1.43G	3.9G
Pretrained Model Website	[56]‡	[57, 58]	n/a	[57–59]	[57–59]	[57–59]

TABLE II
SUMMARY OF POPULAR DNNS [3, 15, 48, 50, 51]. †ACCURACY IS MEASURED BASED ON TOP-5 ERROR ON IMAGENET [14]. ‡THIS VERSION OF LENET-5 HAS 431K WEIGHTS FOR THE FILTERS AND REQUIRES 2.3M MACS PER IMAGE, AND USES RELU RATHER THAN SIGMOID.

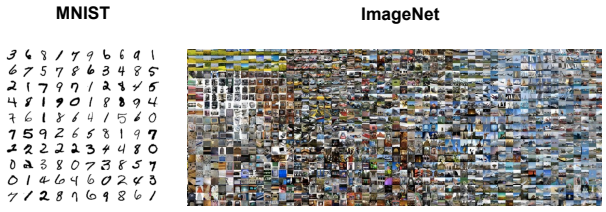


Fig. 16. MNIST (10 classes, 60k training, 10k testing) [62] vs. ImageNet (1000 classes, 1.3M training, 100k testing)[14] dataset.

images (100 per class) and 50,000 validation images (50 per class).

The accuracy of the ImageNet Challenge are reported using two metrics: Top-5 and Top-1 error. Top-5 error means that if any of the top five scoring categories are the correct category, it is counted as a correct classification. The Top-1 requires that the top scoring category be correct. In 2012, the winner of the ImageNet Challenge (AlexNet) was able to achieve an accuracy of 83.6% for the top-5 (which is substantially better than the 73.8% which was second place that year that did not use DNNs); it achieved 61.9% on the top-1 of the validation set. In 2017, the highest accuracy was 97.7% for the top-5.

In summary of the various image classification datasets, it is clear that MNIST is a fairly easy dataset, while ImageNet is a challenging one with a wider coverage of classes. Thus in terms of evaluating the accuracy of a given DNN, it is important to consider that dataset upon which the accuracy is measured.

D. Datasets for Other Tasks

Since the accuracy of the state-of-the-art DNNs are performing better than human-level accuracy on image classification tasks, the ImageNet Challenge has started to focus on more difficult tasks such as single-object localization and object detection. For single-object localization, the target object must be localized and classified (out of 1000 classes). The DNN outputs the top five categories and top five bounding box locations. There is no penalty for identifying an object that is in the image but not included in the ground truth. For object detection, all objects in the image must be localized and classified (out of 200 classes). The bounding box for all objects in these categories must be labeled. Objects that are not labeled are penalized as are duplicated detections.

Beyond ImageNet, there are also other popular image datasets for computer vision tasks. For object detection, there is the PASCAL VOC (2005-2012) dataset that contains 11k images representing 20 classes (27k object instances, 7k of which has detailed segmentation) [68]. For object detection, segmentation and recognition in context, there is the MS COCO dataset with 2.5M labeled instances in 328k images (91 object categories) [69]; compared to ImageNet, COCO has fewer categories but more instances per category, which is useful for precise 2-D localization. COCO also has more labeled instances per image to potentially help with contextual information.

Most recently even larger scale datasets have been made available. For instance, Google has an Open Images dataset with over 9M images [70], spanning 6000 categories. There is also a YouTube dataset with 8M videos (0.5M hours of video) covering 4800 classes [71]. Google also released an audio dataset comprised of 632 audio event classes and a collection of 2M human-labeled 10-second sound clips [72]. These large datasets will be evermore important as DNNs become deeper with more weight parameters to train.

Undoubtedly, both larger datasets and datasets for new domains will serve as important resources for profiling and exploring the efficiency of future DNN engines.

V. HARDWARE FOR DNN PROCESSING

Due to the popularity of DNNs, many recent hardware platforms have special features that target DNN processing. For

instance, the Intel Knights Landing CPU features special vector instructions for deep learning; the Nvidia PASCAL GP100 GPU features 16-bit floating point (FP16) arithmetic support to perform two FP16 operations on a single precision core for faster deep learning computation. Systems have also been built specifically for DNN processing such as Nvidia DGX-1 and Facebook's Big Basin custom DNN server [73]. DNN inference has also been demonstrated on various embedded System-on-Chips (SoC) such as Nvidia Tegra and Samsung Exynos as well as FPGAs. Accordingly, it's important to have a good understanding of how the processing is being performed on these platforms, and how application-specific accelerators can be designed for DNNs for further improvement in throughput and energy efficiency.

The fundamental component of both the CONV and FC layers are the multiply-and-accumulate (MAC) operations, which can be easily parallelized. In order to achieve high performance, highly-parallel compute paradigms are very commonly used, including both temporal and spatial architectures as shown in Fig. 17. The temporal architectures appear mostly in CPUs or GPUs, and employ a variety of techniques to improve parallelism such as vectors (SIMD) or parallel threads (SIMT). Such temporal architecture use a centralized control for a large number of ALUs. These ALUs can only fetch data from the memory hierarchy and cannot communicate directly with each other. In contrast, spatial architectures use dataflow processing, i.e., the ALUs form a processing chain so that they can pass data from one to another directly. Sometimes each ALU can have its own control logic and local memory, called a scratchpad or register file. We refer to the ALU with its own local memory as a processing engine (PE). Spatial architectures are commonly used for DNNs in ASIC and FPGA-based designs. In this section, we will discuss the different design strategies for efficient processing on these different platforms, without any impact on accuracy (i.e., all approaches in this section produce bit-wise identical results); specifically,

- For *temporal* architectures such as CPUs and GPUs, we will discuss how *computational transforms* on the kernel can reduce the number of multiplications to *increase throughput*.
- For *spatial* architectures used in accelerators, we will discuss how *dataflows* can increase data reuse from low cost memories in the memory hierarchy to *reduce energy consumption*.

A. Accelerate Kernel Computation on CPU and GPU Platforms

CPUs and GPUs use parallelizaton techniques such as SIMD or SIMT to perform the MACs in parallel. All the ALUs share the same control and memory (register file). On these platforms, both the FC and CONV layers are often mapped to a matrix multiplication (i.e., the kernel computation). Fig. 18 shows how a matrix multiplication is used for the FC layer. The height of the filter matrix is the number of filters and the width is the number of weights per filter (input channels (C) $\times$ width (W) $\times$ height (H), since $R = W$ and $S = H$ in the FC layer); the height of the input feature maps matrix is the number of activations per input feature map ($C \times W \times H$), and the

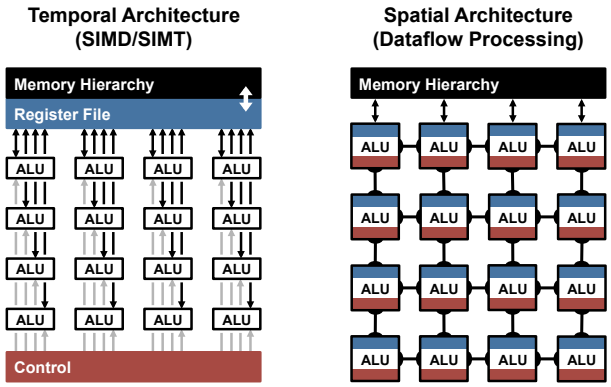
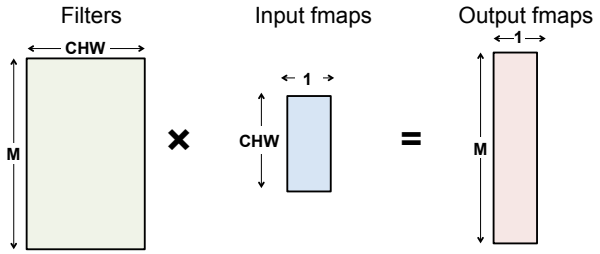
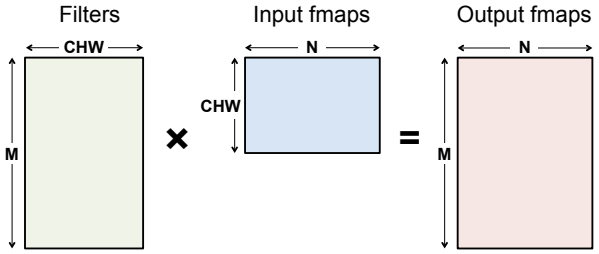


Fig. 17. Highly-parallel compute paradigms.



(a) Matrix Vector multiplication is used when computing a single output feature map from a single input feature map.



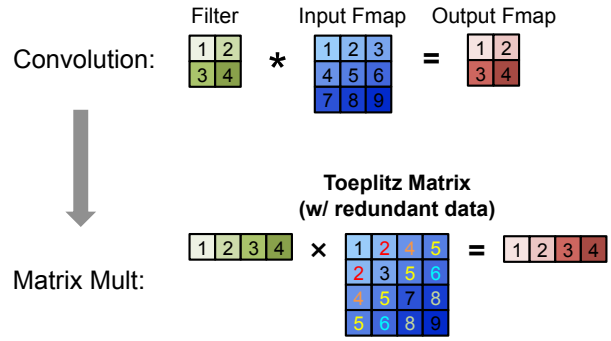
(b) Matrix Multiplications is used when computing N output feature maps from N input feature maps.

Fig. 18. Mapping to matrix multiplication for fully connected layers

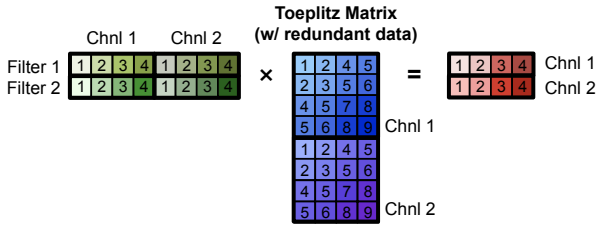
width is the number of input feature maps (one in Fig. 18(a) and N in Fig. 18(b)); finally, the height of the output feature map matrix is the number of channels in the output feature maps (M), and the width is the number of output feature maps (N), where each output feature map of the FC layer has the dimension of $1\times1\times$number of output channels (M).

The CONV layer in a DNN can also be mapped to a matrix multiplication using a relaxed form of the Toeplitz matrix as shown in Fig. 19. The downside for using matrix multiplication for the CONV layers is that there is redundant data in the input feature map matrix as highlighted in Fig. 19(a). This can lead to either inefficiency in storage, or a complex memory access pattern.

There are software libraries designed for CPUs (e.g., OpenBLAS, Intel MKL, etc.) and GPUs (e.g., cuBLAS, cuDNN, etc.) that optimize for matrix multiplications. The matrix multiplication is tiled to the storage hierarchy of these platforms, which are on the order of a few megabytes at the higher levels.



(a) Mapping convolution to Toeplitz matrix



(b) Extend Toeplitz matrix to multiple channels and filters

Fig. 19. Mapping to matrix multiplication for convolutional layers.

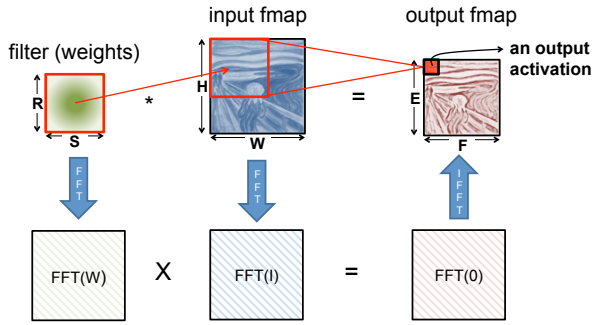


Fig. 20. FFT to accelerate DNN.

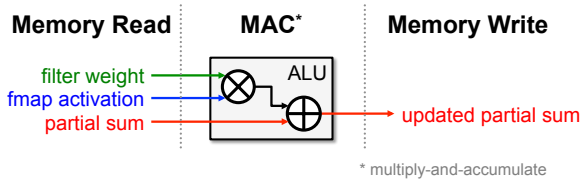


Fig. 21. Read and write access per MAC.

The matrix multiplications on these platforms can be further sped up by applying computational transforms to the data to reduce the number of multiplications, while still giving the same bit-wise result. Often this can come at a cost of increased number of additions and a more irregular data access pattern.

Fast Fourier Transform (FFT) [10, 74] is a well known approach, shown in Fig. 20 that reduces the number of multiplications from $O(N_o^2 N_f^2)$ to $O(N_o^2 log_2 N_o)$, where the output size is $N_o \times N_o$ and the filter size is $N_f \times N_f$. To perform the convolution, we take the FFT of the filter and input feature map, and then perform the multiplication in the frequency domain; we then apply an inverse FFT to the resulting product to recover the output feature map in the spatial domain. However, there are several drawbacks to using FFT: (1) the benefits of FFTs decrease with filter size; (2) the size of the FFT is dictated by the output feature map size which is often much larger than the filter; (3) the coefficients in the frequency domain are complex. As a result, while FFT reduces computation, it requires larger storage capacity and bandwidth. Finally, a popular approach for reducing complexity is to make the weights sparse, which will be discussed in Section VII-B2; using FFTs makes it difficult for this sparsity to be exploited.

Several optimizations can be performed on FFT to make it more effective for DNNs. To reduce the number of operations, the FFT of the filter can be precomputed and stored. In addition, the FFT of the input feature map can be computed once and used to generate multiple channels in the output feature map. Finally, since an image contains only real values, its Fourier Transform is symmetric and this can be exploited to reduce storage and computation cost.

Other approaches include Strassen [75] and Winograd [76], which rearrange the computation such that the number of multiplications reduce from $O(N^3)$ to $O(N^{2.807})$ and by 2.25× for a 3×3 filter, respectively, at the cost of reduced numerical stability, increased storage requirements, and specialized processing depending on the size of the filter.

In practice, different algorithms might be used for different layer shapes and sizes (e.g., FFT for filters greater than 5×5, and Winograd for filters 3×3 and below). Existing platform libraries, such as MKL and cuDNN, dynamically chose the appropriate algorithm for a given shape and size [77, 78].

B. Energy-Efficient Dataflow for Accelerators

For DNNs, the bottleneck for processing is in the memory access. Each MAC requires three memory reads (for filter weight, fmap activation, and partial sum) and one memory write (for the updated partial sum) as shown in Fig. 21. In the worst case, all of the memory accesses have to go through the off-chip DRAM, which will severely impact both throughput and energy efficiency. For example, in AlexNet, to support its 724M MACs, nearly 3000M DRAM accesses will be required. Furthermore, DRAM accesses require up to several orders of magnitude higher energy than computation [79].

Accelerators, such as spatial architectures as shown in Fig. 17, provide an opportunity to reduce the energy cost of data movement by introducing several levels of local memory hierarchy with different energy cost as shown in Fig. 22. This includes a large global buffer with a size of several hundred kilobytes that connects to DRAM, an inter-PE network that can pass data directly between the ALUs, and a register file (RF) within each processing element (PE) with a size of a few kilobytes or less. The multiple levels of memory hierarchy help to improve energy efficiency by providing low-cost data accesses. For example, fetching the data from the RF or neighbor PEs is going to cost 1 or 2 orders of magnitude lower energy than from DRAM.

Accelerators can be designed to support specialized processing dataflows that leverage this memory hierarchy. The dataflow

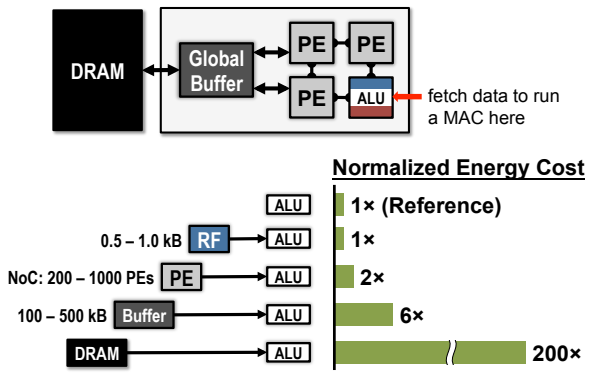


Fig. 22. Memory hierarchy and data movement energy [80].

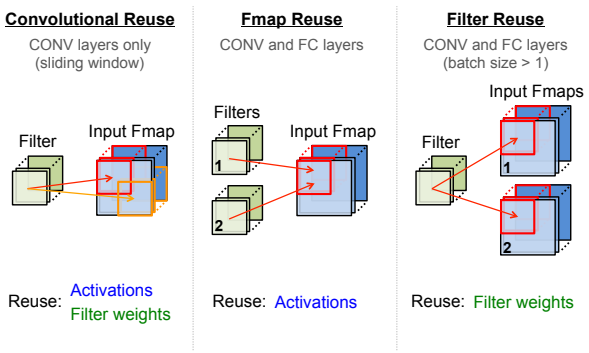


Fig. 23. Data reuse opportunities in DNNs [80].

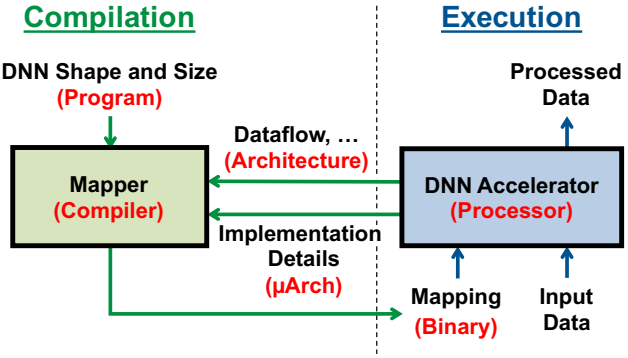


Fig. 24. An analogy between the operation of DNN accelerators (texts in black) and that of general-purpose processors (texts in red). Figure adopted from [81].

decides what data gets read into which level of the memory hierarchy and when are they getting processed. Since there is no randomness in the processing of DNNs, it is possible to design a fixed dataflow that can adapt to the DNN shapes and sizes and optimize for the best energy efficiency. The optimized dataflow minimizes access from the more energy consuming levels of the memory hierarchy. Large memories that can store a significant amount of data consume more energy than smaller memories. For instance, DRAM can store gigabytes of data, but consumes two orders of magnitude higher energy per access than a small on-chip memory of a few kilobytes. Thus, every time a piece of data is moved from an expensive level to a lower cost level in terms of energy, we want to reuse that piece of data as much as possible to minimize subsequent accesses to the expensive levels. The challenge, however, is that the storage capacity of these low cost memories are limited. Thus we need to explore different dataflows that maximize reuse under these constraints.

For DNNs, we investigate dataflows that exploit three forms of input data reuse (convolutional, feature map and filter) as shown in Fig. 23. For convolutional reuse, the same input feature map activations and filter weights are used within a given channel, just in different combinations for different weighted sums. For feature map reuse, multiple filters are applied to the same feature map, so the input feature map activations are used multiple times across filters. Finally, for filter reuse, when multiple input feature maps are processed at once (referred to as a batch), the same filter weights are used multiple times across input features maps.

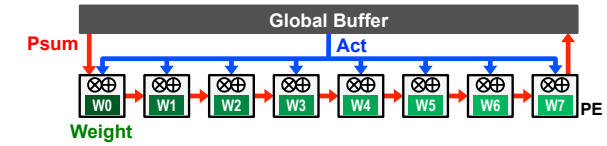
If we can harness the three types of data reuse by storing the data in the local memory hierarchy and accessing them multiple times without going back to the DRAM, it can save a significant amount of DRAM accesses. For example, in AlexNet, the number of DRAM reads can be reduced by up to 500× in the CONV layers. The local memory can also be used for partial sum accumulation, so they do not have to reach DRAM. In the best case, if all data reuse and accumulation can be achieved by the local memory hierarchy, the 3000M DRAM accesses in AlexNet can be reduced to only 61M.

The operation of DNN accelerators is analogous to that of general-purpose processors as illustrated in Fig. 24 [81]. In conventional computer systems, the compiler translates the program into machine-readable binary codes for execution given the hardware architecture (e.g., x86 or ARM); in the processing of DNNs, the mapper translates the DNN shape and size into a hardware-compatible computation mapping for execution given the dataflow. While the compiler usually optimizes for performance, the mapper optimizes for energy efficiency.

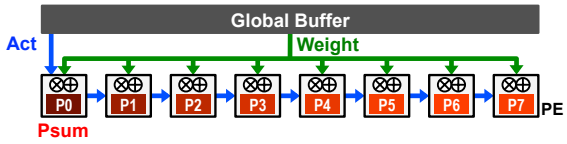
The following taxonomy (Fig. 25) can be used to classify the DNN dataflows in recent works [82–93] based on their data handling characteristics [80]:

1) Weight stationary (WS): The weight stationary dataflow is designed to minimize the energy consumption of reading weights by maximizing the accesses of weights from the register file (RF) at the PE (Fig. 25(a)). Each weight is read from DRAM into the RF of each PE and stays stationary for further accesses. The processing runs as many MACs that use the same weight as possible while the weight is present in the RF; it maximizes convolutional and filter reuse of weights. The inputs and partial sums must move through the spatial array and global buffer. The input fmap activations are broadcast to all PEs and then the partial sums are spatially accumulated across the PE array.

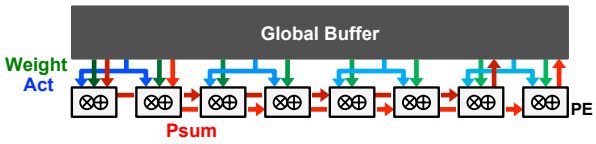
One example of previous work that implement weight stationary dataflow is nn-X, or neuFlow [85], which uses eight 2-D convolution engines for processing a 10×10 filter. There are total 100 MAC units, i.e. PEs, per engine with each PE having a weight that stays stationary for processing. The



(a) Weight Stationary



(b) Output Stationary



(c) No Local Reuse

Fig. 25. Dataflows for DNNs [80].

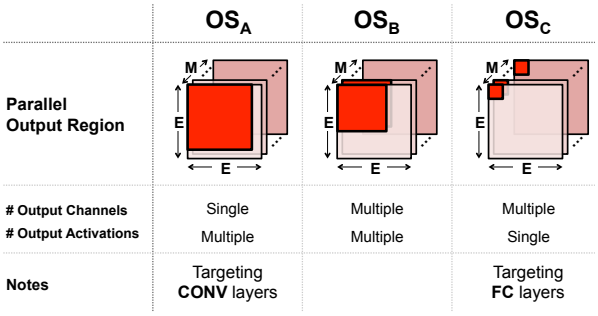


Fig. 26. Variations of output stationary [80].

input fmap activations are broadcast to all MAC units and the partial sums are accumulated across the MAC units. In order to accumulate the partial sums correctly, additional delay storage elements are required, which are counted into the required size of local storage. Other weight stationary examples are found in [82–84, 86, 87].

2) Output stationary (OS): The output stationary dataflow is designed to minimize the energy consumption of reading and writing the partial sums (Fig. 25(b)). It keeps the accumulation of partial sums for the same output activation value local in the RF. In order to keep the accumulation of partial sums stationary in the RF, one common implementation is to stream the input activations across the PE array and broadcast the weight to all PEs in the array.

One example that implements the output stationary dataflow is ShiDianNao [89], where each PE handles the processing for each output activation value by fetching the corresponding input activations from neighboring PEs. The PE array implements dedicated networks to pass data horizontally and vertically. Each PE also has data delay registers to keep data around for the required amount of cycles. At the system level, the global buffer streams the input activations and broadcasts the weights into the PE array. The partial sums are accumulated inside each PE and then get streamed out back to the global buffer. Other examples of output stationary are found in [88, 90].

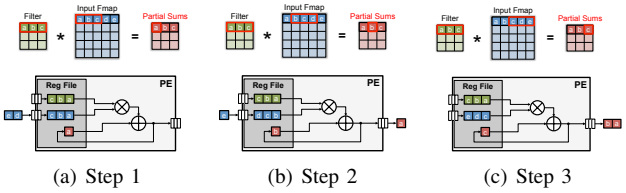
There are multiple possible variants of output stationary as shown in Fig. 26 since the output activations that get processed at the same time can come from different dimensions. For example, the variant OS_A targets the processing of CONV layers, and therefore focuses on the processing of output activations from the same channel at a time in order to maximize data reuse opportunities. The variant OS_C targets the processing of FC layers, and focuses on generating output activations from all different channels, since each channel only has one output activation. The variant OS_B is something in between OS_A and OS_C. Example of variants OS_A, OS_B, and OS_C are [89], [88], and [90], respectively.

3) No local reuse (NLR): While small register files are efficient in terms of energy (pJ/bit), they are inefficient in terms of area (μm^2/bit). In order to maximize the storage capacity, and minimize the off-chip memory bandwidth, no local storage is allocated to the PE and instead all that area is allocated to the global buffer to increase its capacity (Fig. 25(c)). The no local reuse dataflow differs from the previous dataflows in that nothing stays stationary inside the PE array. As a result, there will be increased traffic on the spatial array and to the global buffer for all data types. Specifically, it has to multicast the activations, single-cast the filter weights, and then spatially accumulate the partial sums across the PE array.

In an example of the no local reuse dataflow from UCLA [91], the filter weights and input activations are read from the global buffer, processed by the MAC units with custom adder trees that can complete the accumulation in a single cycle, and the resulting partial sums or output activations are then put back to the global buffer. Another example is DianNao [92], which also reads input activations and filter weights from the buffer, and processes them through the MAC units with custom adder trees. However, DianNao implements specialized registers to keep the partial sums in the PE array, which helps to further reduce the energy consumption of accessing partial sums. Another example of no local reuse dataflow is found in [93].

4) Row stationary (RS): A row stationary dataflow is proposed in [80], which aims to maximize the reuse and accumulation at the RF level for *all* types of data (weights, pixels, partial sums) for the overall energy efficiency. This differs from WS or OS dataflows, which optimize for only weights and partial sums, respectively.

The row stationary dataflow assigns the processing of a 1-D row convolution into each PE for processing as shown in Fig. 27. It keeps the row of filter weights stationary inside the RF of the PE and then streams the input activations into the PE. The PE does the MACs for each sliding window at a time, which uses just one memory space for the accumulation of partial sums. Since there are overlaps of input activations between different sliding windows, the input activations can then be kept in the RF and get reused. By going through all the sliding windows in the row, it completes the 1-D convolution and maximize the data reuse and local accumulation of data in this row.



(a) Step 1 (b) Step 2 (c) Step 3

Fig. 27. 1-D Convolutional reuse within PE for Row Stationary Dataflow [80].

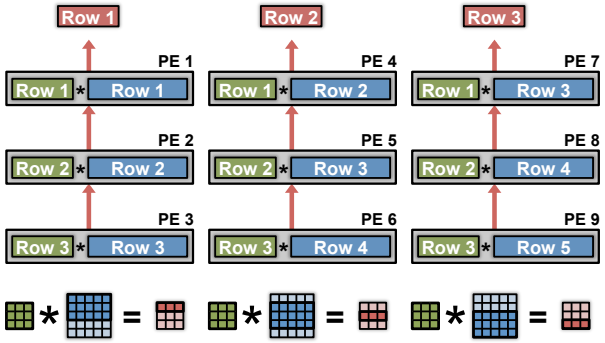


Fig. 28. 2-D convolutional reuse within spatial array for Row Stationary Dataflow [80].

With each PE processing a 1-D convolution, multiple PEs can be aggregated to complete the 2-D convolution as shown in Fig. 28. For example, to generate the first row of output activations with a filter having three rows, three 1-D convolutions are required. Therefore, we can use three PEs in a column, each running one of the three 1-D convolutions. The partial sums are further accumulated vertically across the three PEs to generate the first output row. To generate the second row of output, we use another column of PEs, where three rows of input activations are shifted down by one row, and use the same rows of filters to perform the three 1-D convolutions. Additional columns of PEs are added until all rows of the output are completed (i.e., the number of PE columns equals the number of output rows).

This 2-D array of PEs enables other forms of reuse to reduce accesses to the more expensive global buffer. For example, each filter row is reused across multiple PEs horizontally. Each row of input activations is reused across multiple PEs diagonally. And each row of partial sums are further accumulated across the PEs vertically. Therefore, 2-D convolutional data reuse and accumulation are maximized inside the 2-D PE array.

To address the high-dimensional convolution of the CONV layer (i.e., multiple fmaps, filters, and channels), multiple rows can be mapped onto the same PE as shown in Fig. 29. The 2-D convolution is mapped to a set of PEs, and the additional dimensions are handled by interleaving or concatenating the additional data. For filter reuse within the PE, different rows of fmaps are concatenated and run through the same PE as a 1-D convolution. For input fmap reuse within the PE, different filter rows are interleaved and run through the same PE as a 1-D convolution. Finally, to increase local partial sum accumulation within the PE, filter rows and fmap rows from

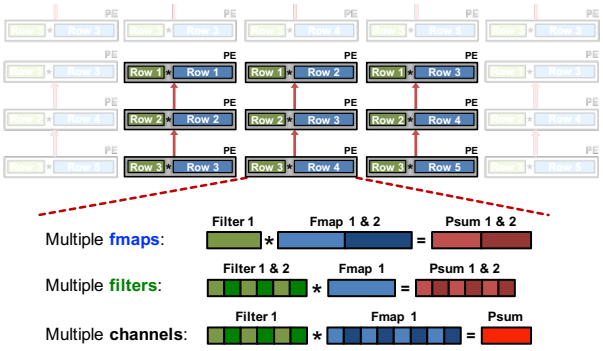


Fig. 29. Multiple rows of different input feature maps, filters and channels are mapped to same PE within array for additional reuse in the Row Stationary Dataflow [80].

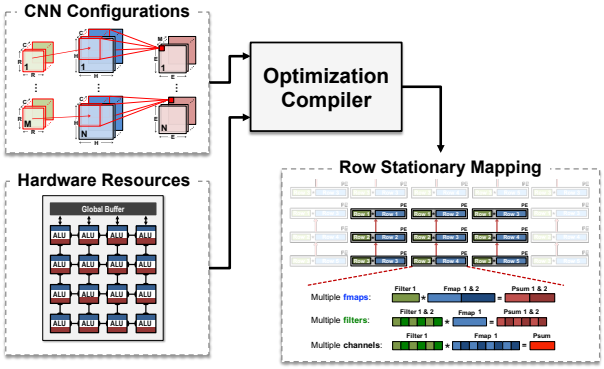


Fig. 30. Mapping optimization takes in hardware and DNNs shape constraints to determine optimal energy dataflow [80].

different channels are interleaved, and run through the same PE as a 1-D convolution. The partial sums from different channels then naturally get accumulated inside the PE.

The number of filters, channels, and fmaps that can be processed at the same time is programmable, and there exists an optimal mapping for the best energy efficiency, which depends on the shape configuration of the DNN as well as the hardware resources provided, e.g., the number of PEs and the size of the memory in the hierarchy. Since all of the variables are known before runtime, it is possible to build a compiler (i.e., mapper) to perform this optimization off-line to configure the hardware for different mappings of the RS dataflow for different DNNs as shown in Fig. 30.

One example that implements the row stationary dataflow is Eyeriss [94]. It consists of a 14×12 PE array, a 108KB global buffer, ReLU and fmap compression units as shown in Fig. 31. The chip communicates with the off-chip DRAM using a 64-bit bidirectional data bus to fetch data into the global buffer. The global buffer then streams the data into the PE array for processing.

In order to support the RS dataflow, two problems need to be solved in the hardware design. First, how can the fixed-size PE array accommodate different layer shapes? Second, although the data will be passed in a very specific pattern, it still changes with different shape configurations. How can the fixed design

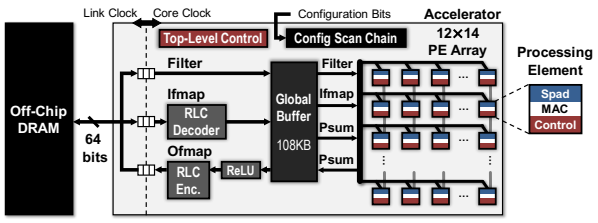


Fig. 31. Eyeriss DNN accelerator [94].

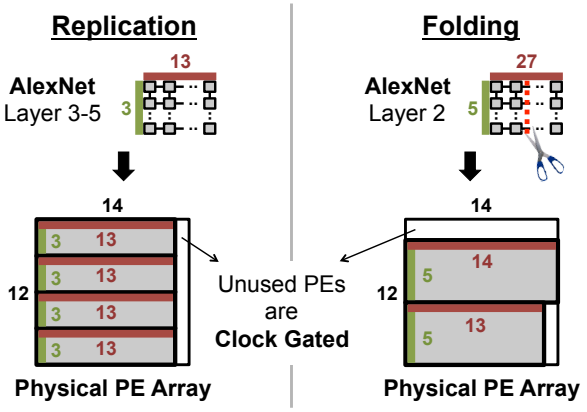


Fig. 32. Mapping uses replication and folding to maximized utilization of PE array [94].

pass data in different patterns?

Two mapping strategies can be used to solve the first problem as shown in Fig. 32. First, replication can be used to map shapes that do not use up the entire PE array. For example, in the third to fifth layers of AlexNet, each 2-D convolution only uses a 13×3 PE array. This structure is then replicated four times, and runs different channels and filters in each replication. The second strategy is called folding. For example, in the second layer of AlexNet, it requires a 27×5 PE array to complete the 2-D convolution. In order to fit it into the 14×12 physical PE array, it is folded into two parts, 14×5 and 13×5, and each are vertically mapped into the physical PE array. Since not all PEs are used by the mapping, the unused PEs can be clock gated to save energy consumption.

A custom multicast network is used to solve the second problem about flexible data delivery. The simplest way to pass data to multiple destinations is to broadcast the data to all PEs and let each PE decide if it has to process the data or not. However, it is not very energy efficient especially when the size of PE array is large. Instead, a multicast network is used to send data to only the places where it is needed.

5) Energy comparison of different dataflows: To evaluate and compare different dataflows, the same total hardware area and number of PEs (256) are used in the simulation of a spatial architecture for all dataflows. The local memory (register file) at each processing element (PE) is on the order of 0.5 – 1.0kB and a shared memory (global buffer) is on the order of 100 – 500kB. The sizes of these memories are selected to be comparable to a typical accelerator for multimedia processing, such as video coding [95]. The memory sizes are further adjusted for the needs of each dataflow under the same area constraint. For example, since the no local reuse dataflow does not require any RF in PE, it is allocated with a much larger global buffer. The simulation uses the layer configurations from AlexNet with a batch size of 16. The simulation also takes into account the fact that accessing different levels of the memory hierarchy requires different energy cost.

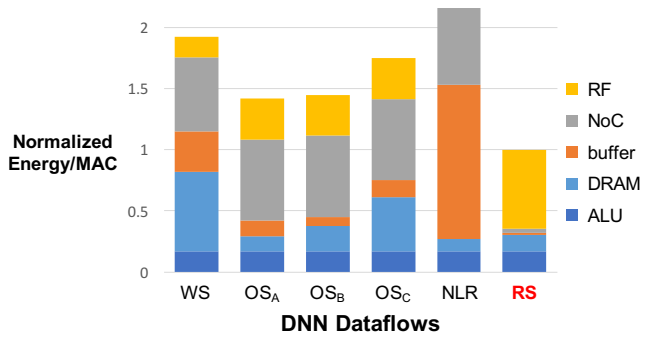
Fig. 33 compares the chip and DRAM energy consumption of each dataflow for the CONV layers of AlexNet with a batch size of 16. The WS and OS dataflows have the lowest energy consumption for accessing weights and partial sums, respectively. However, the RS dataflow has the lowest total energy consumption since it optimizes for the overall energy efficiency instead of only for a certain data type.

Fig. 33(a) shows the same results with breakdown in terms of memory hierarchy. The RS dataflow consumes the most energy in the RF, since by design most of the accesses have been moved to the lowest level of the memory hierarchy. This helps to achieve the lowest total energy consumption since RF has the lowest energy per access. The NLR dataflow has the lowest energy consumption at the DRAM level, since it has a much larger global buffer and thus higher on-chip storage capacity compared to others. However, most of the data accesses in the NLR dataflow is from the global buffer, which still has a relatively large energy consumption per access compared to accessing data from RF or inside the PE array. As a result, the overall energy consumption of the NLR dataflow is still fairly high. Overall, RS dataflow uses 1.4× to 2.5× lower energy than other dataflows.

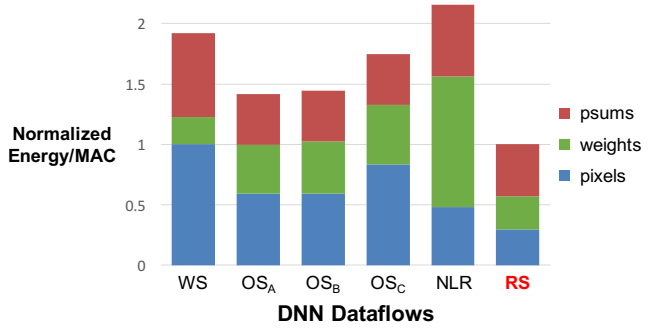
Fig. 34 shows the energy efficiency between different dataflows in the FC layers of AlexNet with a batch size of 16. Since there is not as much data reuse in the FC layers as in the CONV layers, all dataflows spend a significant amount of energy on reading weights. However, RS dataflow still has the lowest energy consumption because it optimizes for the energy of accessing input activations and partial sums. For the OS dataflows, OS_C now consumes lower energy than OS_A since it is designed for the FC layers. Overall, RS still consumes 1.3× lower energy compared to other dataflows at the batch size of 16.

Fig. 35 shows the RS dataflow design with energy breakdown in terms of different layers of AlexNet. In the CONV layers, the energy is mostly consumed by the RF, while in the FC layers, the energy is mostly consumed by DRAM. However, most of the energy is consumed by the CONV layers, which takes around 80% of the energy. As recent DNN models go deeper with more CONV layers, the ratio between number of CONV and FC layers only gets larger. Therefore, moving forward, significant effort should be placed on energy optimizations for CONV layers.

Finally, up until now, we have been looking at architectures with relatively limited storage on the order of a few hundred kilobytes. With much larger storage on the order of a few megabytes, additional dataflows can be considered. For example, Fused-Layer looks at dataflow optimizations across layers [96].



(a) Energy breakdown across memory hierarchy



(b) Energy breakdown across data type

Fig. 33. Comparison of energy efficiency between different dataflows in the CONV layers of AlexNet with a batch size of 16 [3]: (a) breakdown in terms of storage levels and ALU, (b) breakdown in terms of data types. OS$_A$, OS$_B$ and OS$_C$ are three variants of the OS dataflow that are commonly seen in different implementations [80].

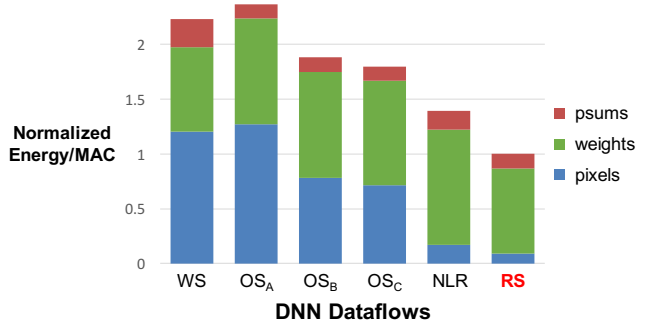


Fig. 34. Comparison of energy efficiency between different dataflows in the FC layers of AlexNet with a batch size of 16 [80].

VI. Near-Data Processing

The previous section highlighted that data movement dominates energy consumption. While spatial architectures distribute the on-chip memory such that it is closer to the computation (e.g., into the PE), there have also been efforts to bring the off-chip high density memory closer to the computation or to integrate the computation into the memory itself; the latter is often referred to as *processing-in-memory* or *logic-in-memory*. In embedded systems, there have also been efforts to bring the computation into the sensor where the data is first collected.

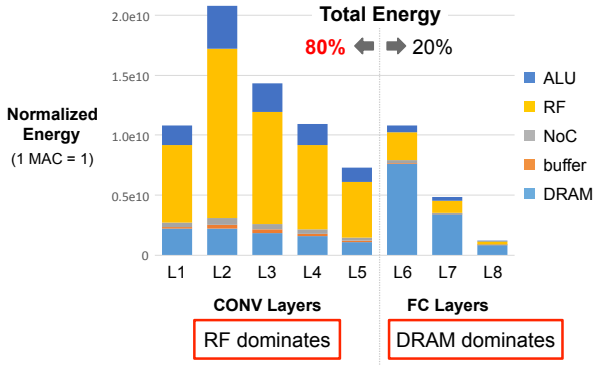


Fig. 35. Energy breakdown across layers of the AlexNet [80]. RF energy dominates in convolutional layers. DRAM energy dominates in the fully connected layer. Convolutional layer dominate energy consumption.

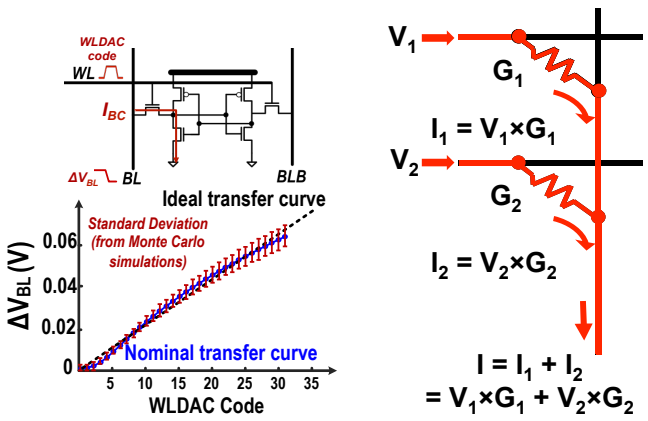
In this section, we will discuss how moving compute and data closer to reduce data movement (i.e., near-data processing) can be achieved using mixed-signal circuit design and advanced memory technologies.

Many of these works use analog processing which has the drawback of increased sensitivity to circuit and device non-idealities. Consequentially, the computation is often performed at reduced precision, which can be accounted for during the training of the DNNs using the techniques discussed in Section VII. Another factor to take into consideration is that DNNs are often trained in the digital domain; thus for analog processing, there is an additional overhead cost for analog-to-digital conversion (ADC) and digital-to-analog conversion (DAC).

A. DRAM

Advanced memory technology can reduce the access energy for high density memories such as DRAMs. For instance, *embedded DRAM (eDRAM)* brings high density memory on-chip to avoid the high energy cost of switching off-chip capacitance [97]; eDRAM is 2.85× higher density than SRAM and 321× more energy efficient than DRAM (DDR3) [93]. eDRAM also offers higher bandwidth and lower latency compared to DRAM. In DNN processing, eDRAM can be used to store tens of megabytes of weights and activations on-chip to avoid off-chip access, as demonstrated in DaDianNao [93]. The downside of eDRAM is that it has lower density than off-chip DRAM and can increase the cost of the chip.

Rather than integrating DRAM into the chip itself, the DRAM can also be stacked on top of the chip using through silicon vias (TSV). This technology is often referred to as *3-D memory*, and has been commercialized in the form of Hybrid Memory Cube (HMC) [98] and High Bandwidth Memory (HBM) [99]. 3-D memory delivers an order of magnitude higher bandwidth and reduces access energy by up to 5× relative to existing 2-D DRAMs, as TSV have lower capacitance than typical off-chip interconnects. Recent works have explored the use of HMC for efficient DNN processing in a variety of ways. For instance, Neurocube [100] integrates SIMD processors into the logic die of the HMC to bring the memory and computation



(a) Multiplication performed by bit-cell (Figure from [102])

(b) G_i is conductance of resistive memory (Figure from [104])

Fig. 36. Analog computation by (a) SRAM bit-cell and (b) non-volatile resistive memory.

closer together. Tetris [101] explores the use of HMC with the Eyeriss spatial architecture and row stationary dataflow. It proposes allocating more area to computation than on-chip memory (i.e., larger PE array and smaller global buffer) in order to exploit the low energy and high throughput properties of the HMC. It also adapts the dataflow to account for the HMC memory and smaller on-chip memory. Tetris achieves a 1.5× reduction in energy consumption and 4.1× increase in throughput over a baseline system with conventional 2-D DRAM.

B. SRAM

Rather than bringing the memory near the compute, recent work has also investigated bringing the compute into the memory. For instance, the multiply and accumulate operation can be directly integrated into the bit-cells of an SRAM array [102], as shown in Fig. 36(a). In this work, a 5-bit DAC is used to drive the word line (WL) to an analog voltage that represents the feature vector, while the bit-cells store the binary weights ±1. The bit-cell current (I_{BC}) is effectively a product of the value of the feature vector and the value of the weight stored in the bit-cell; the currents from the bit-cells within a column add together to discharge the bitline (V_{BL}). This approach gives 12× energy savings compared to reading the 1-bit weights from the SRAM and performing the computation separately. To counter circuit non-idealities, the DAC accounts for the non-linear bit-line discharge with respect to the WL voltage, and boosting is used to combine the weak classifiers that are susceptible to device variations to form a strong classifier [103].

C. Non-volatile Resistive Memories

The multiply and accumulate operation can also be directly integrated into advanced *non-volatile* high density memories by using them as programmable resistive elements, commonly referred to as *memristors* [105]. Specifically, a multiplication is performed with the resistor's conductance as the weight, the voltage as the input, and the current as the output as shown in Fig. 36(b). The addition is done by summing the currents of different memristors with Kirchhoff's current law. This is the ultimate form of a weight stationary dataflow, as the weights are always held in place. The advantages of this approach include reduced energy consumption since the computation is embedded within memory which reduces data movement, and increased density since memory and computation can be densely packed with a similar density to DRAM [106].[8]

There are several popular candidates for non-volatile resistive memory devices including phase change memory (PCM), resistive RAM (RRAM or ReRAM), conductive bridge RAM (CBRAM), and spin transfer torque magnetic RAM (STT-MRAM) [107]. These devices have different trade-offs in terms of endurance (i.e., how many times it can be written), retention time, write current, density (i.e., cell size), variations and speed.

Processing with non-volatile resistive memories has several drawbacks as described in [108]. First, it suffers from the reduced precision and ADC/DAC overhead of analog processing described earlier. Second, the array size is limited by the wires that connect the resistive devices; specifically, wire energy dominates for large arrays (e.g., 1k×1k), and the IR drop along wire can degrade the read accuracy. Third, the write energy to program the resistive devices can be costly, in some cases requiring multiple pulses. Finally, the resistive devices can also suffer from device-to-device and cycle-to-cycle variations with non-linear conductance across the conductance range.

There have been several recent works that explore the use of memristors for DNNs. ISAAC [104] replaces the eDRAM in DaDianNao with memristors. To address the limited precision support, ISAAC computes a 16-bit dot product operation with 8 memristors each storing 2-bits; a 1-bit×2-bit multiplication is performed at each memristor, where a 16-bit input requires 16 cycles to complete. In other words, the ISAAC architecture trades off area and time for increased precision. Finally, ISAAC arranges its 25.1M memristors in a hierarchical structure to avoid issues with large arrays. PRIME [109] also replaces the DRAM main memory with memristors; specifically, it uses 256×256 memristor arrays that can be configured for 4-bit multi-level cell computation or 1-bit single level cell storage. It should be noted that results from ISAAC and PRIME are obtained from simulations. The task of actually fabricating large memristors arrays is still very much a research challenge; for instance, [110] uses a fabricated 12×12 memristor array to demonstrate a linear classifier.

D. Sensors

In certain applications, such as image processing, the data movement from the sensor itself can account for a significant portion of the system energy consumption. Thus there has also been research on performing the computation as close as possible to the sensor. In particular, much of the work focuses on moving the computation into the analog domain to avoid using the ADC within the sensor, which accounts for a significant portion of the sensor power. However, as mentioned

[8]The resistive devices can be inserted between the cross-point of two wires and in certain cases can avoid the need for an access transistor.

earlier, lower precision is required for analog computation due to circuit non-idealities.

In [111], the matrix multiplication is integrated into the ADC, where the most significant bits of the multiplications are performed using switched capacitors in an 8-bit successive approximation format. This is extended in [112] to not only perform the multiplications, but also the accumulations in the analog domain. In this work, it is assumed that 3-bits and 6-bits are sufficient to represent the weights and activations, respectively. This reduces the number of ADC conversions in the sensor by 21×. RedEye [113] takes this approach even further by performing the entire convolution layer (including convolution, max pooling and quantization) in the analog domain at the sensor. It should be noted that [111] and [112] report measured results from fabricated test chips, while results in [113] are from simulations.

It is also feasible to embed the computation not just before the ADC, but into the sensor itself. For instance, in [114] an Angle Sensitive Pixels sensor is used to compute the gradient of the input, which along with compression, reduces the data movement from the sensor by 10×. In addition, since the first layer of the DNN often outputs a gradient-like feature map, it maybe possible to skip the computations in the first layer, which further reduces energy consumption as discussed in [115, 116].

VII. Co-design of DNN models and Hardware

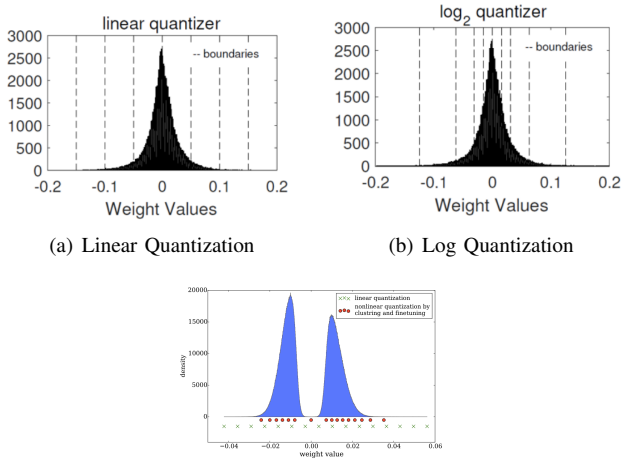
In earlier work, the DNN models were designed to maximize accuracy without much consideration of the implementation complexity. However, this can lead to designs that are challenging to implement and deploy. To address this, recent work has shown that DNN models and hardware can be co-designed to jointly maximize accuracy and throughput, while minimizing energy and cost, which increases the likelihood of adoption. In this section, we will highlight various efforts that have been made towards the co-design of DNN models and hardware. Note that unlike Section V, the techniques discussed in this section can affect the accuracy; thus, the goal is to not only substantially reduce energy consumption and increase throughput, but also to minimize any degradation in accuracy.

The co-design approaches can be loosely grouped into the following categories:

- *Reduce precision of operations and operands.* This includes going from floating point to fixed point, reducing the bitwidth, non-linear quantization and weight sharing.
- *Reduce number of operations and model size.* This includes techniques such as compression, pruning and compact network architectures.

A. Reduce Precision

Quantization involves mapping data to a smaller set of quantization levels. The ultimate goal is to minimize the error between the reconstructed data from the quantization levels and the original data. The number of quantization levels reflects the *precision* and ultimately the number of bits required to represent the data (usually log_2 of the number of levels); thus, *reduced precision* refers to reducing the number of levels, and thus the number of bits. The benefits of reduced precision include reduced storage cost and/or reduced computation requirements.



(a) Linear Quantization (b) Log Quantization

(c) Non-Linear Quantization

Fig. 37. Various methods of quantization (Figures from [117, 118]).

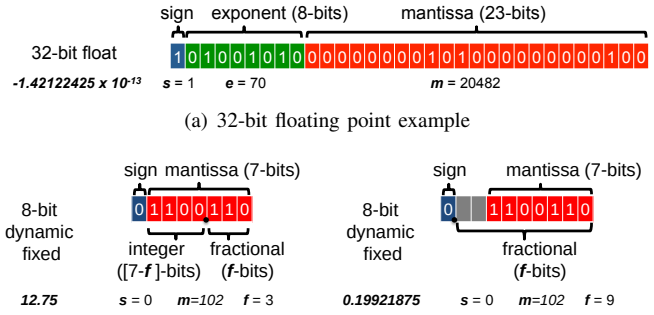
There are several ways to map the data to quantization levels. The simplest method is a linear mapping with uniform distance between each quantization level (Fig. 37(a)). Another approach is to use a simple mapping function such as a *log function* (Fig. 37(b)) where the distance between the levels varies; this mapping can often be implemented with simple logic such as a shift. Alternatively, a more complex mapping function can be used where the quantization levels are determined or learned from the data (Fig. 37(c)), e.g., using k-means clustering; for this approach, the mapping is usually implemented with a look up table.

Finally, the quantization can be fixed (i.e., the same method of quantization is used for all data types and layers, filters, and channels in the network); or it can be variable (i.e., different methods of quantization can be used for weights and activations, and different layers, filters, and channels in the network).

Reduced precision research initially focused on reducing the precision of the weights rather than the activations, since weights directly increase the storage capacity requirement, while the impact of activations on storage capacity depends on the network architecture and dataflow. However, more recent works have also started to look at the impact of quantization on activations. Most reduced precision research also focuses on reducing the precision for inference rather than training (with some exceptions [88, 119, 120]) due to the sensitivity of the gradients to quantization.

The key techniques used in recent work to reduce precision are summarized in Table III; both linear and non-linear quantization applied to weights and activations are explored. The impact on accuracy is reported relative to a baseline precision of 32-bit floating point, which is the default precision used on platforms such as GPUs and CPUs.

1) Linear quantization: The first step of reducing precision is usually to convert values and operations from floating point to fixed point. A 32-bit floating point number, as shown in Fig. 38(a), is represented by $(-1)^s \times m \times 2^{(e-127)}$, where s



(a) 32-bit floating point example

(b) 8-bit dynamic fixed point examples

Fig. 38. Various methods of number representations.

is the sign bit, e is the 8-bit exponent, and m is the 23-bit mantissa, and covers the range of 10^{-38} to 10^{38}.

An N-bit fixed point number is represented by $(-1)^s \times m \times 2^{-f}$, where s is the sign bit, m is the (N-1)-bit mantissa, and f determines the location of the decimal point and acts as a scale factor. For instance, for an 8-bit integer, when $f = 0$, the dynamic range is -128 to 127, whereas when $f = 10$, the dynamic range is -0.125 to 0.124023438. *Dynamic* fixed point representation allows f to vary based on the desired dynamic range as shown in Fig. 38(b). This is useful for DNNs, since the dynamic range of the weights and activations can be quite different. In addition, the dynamic range can also vary across layers and layer types (e.g., convolutional vs. fully connected). Using dynamic fixed point, the bitwidth can be reduced to 8 bits for the weights and 10 bits for the activations without any fine-tuning of the weights [121]; with fine-tuning, both weights and activations can reach 8-bits [122].

Using 8-bit fixed point has the following impact on energy and area [79]:

- An 8-bit fixed point add consumes 3.3× less energy (3.8× less area) than a 32-bit fixed point add, and 30× less energy (116× less area) than a 32-bit floating point add. The energy and area of a fixed-point add scales approximately linearly with the number of bits.
- An 8-bit fixed point multiply consumes 15.5× less energy (12.4× less area) than a 32-bit fixed point multiply, and 18.5× less energy (27.5× less area) than a 32-bit floating point multiply. The energy and area of a fixed-point multiply scales approximately quadratically with the number of bits.

Reducing the precision also reduces the energy and area cost for storage, which is important since memory access and data movement dominate energy consumption as described earlier. The energy and area of the memory scale approximately linearly with number of bits. It should be noted, however, that changing from floating point to fixed point, without reducing bit-width, does not reduce the energy or area cost of the memory.

For completeness, it should be noted that the precision of the internal values of a fixed-point multiply and accumulate (MAC) operation are typically higher than the weights and activations. To guarantee no precision loss, weights and input activations with N-bit fixed-point precision would require an N-bit×N-bit multiplication which generates a 2N-bit output

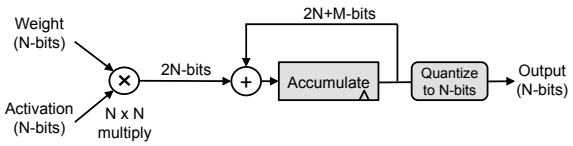


Fig. 39. Reducing the precision of multiply and accumulate (MAC).

product; that output would need to be accumulated with 2N+M-bit precision, where M is determined based on the largest filter size $log_2(C \times R \times S$ from Fig. 9(b)), which is in the range of 10 to 16 bits for the popular DNNs described in Section III-B. After accumulation, the precision of the final output activation is typically reduced to N-bits [88, 121], as shown in Fig. 39. The reduced output precision does not have a significant impact on accuracy if the distribution of the weights and activations are centered near zero such that the accumulation would not move only in one direction; this is particularly true when batch normalization is used.

The reduced precision is not only explored in research, but has been used in recent commercial platforms for DNN processing. For instance, Google's Tensor Processing Unit (TPU) which was announced in May 2016, was designed for 8-bit integer arithmetic [123]. Similarly, Nvidia's PASCAL GPU, which was announced in April 2016, also has 8-bit integer instructions for deep learning inference [124]. In general purpose platforms such as CPUs and GPUs, the main benefit of using 8-bit computation is an increase in throughput, as four 8-bit operations rather than one 32-bit operation can be performed for a given clock cycle.

While general purpose platforms usually support 8-bit, 16-bit and/or 32-bit operations, it has been shown that the minimum bit precision for DNNs can actually vary in a more fine grained manner. For instance, the weight and activation precision can vary between 4 and 9 bits for AlexNet across different layers without significant impact on accuracy (i.e., a change of less than 1%) [125, 126]. This fine-grained variation can be exploited for increased throughput or reduced energy consumption with specialized hardware. For instance, if bit-serial processing is used, where the number of clock cycles to complete an operation is proportional to the bitwidth, adapting to fine-grain variations in bit precision can result in a 2.24× speed up versus 16-bits [125]. Alternatively, a multiplier can be designed such that its critical path reduces based on the bit precision as fewer adders are needed to resolve the product; this can be combined with voltage scaling for a 2.56× energy savings versus 16-bits [126]. While these bit scaling results are reported relative to 16-bit, it would be interesting to see their impact relative to the maximum precision required across layers (i.e., 9-bits for [125, 126]).

The precision can be reduced even more aggressively to a single bit; this area of research is often referred to as *binary nets*. BinaryConnect (BC) [127] introduced the concept of binary weights (i.e., -1 and 1), where using a binary weight reduced the multiplication in the MAC to addition and subtraction only. This was later extended in Binarized Neural Networks (BNN) [128] that uses binary weights *and* activations, which

reduces the MAC to an XNOR. However, BC and BNN have an accuracy loss of 19% and 29.8%, respectively [129].

In order to reduce this accuracy loss, Binary Weight Nets (BWN) and XNOR-Nets introduced several significant modifications to the DNN processing [129]. This includes multiplying the outputs with a scale factor to recover the dynamic range (i.e., the weights effectively become -w and w, where w is the average of the absolute values of the weights in the filter)[9], keeping the first and last layers at 32-bit floating point precision, and performing normalization before convolution to reduce the dynamic range of the activations. With these changes, BWN reduced the accuracy loss to 0.8%, while XNOR-Nets reduced the loss to 11%. The loss of XNOR-Net can be further reduced by increasing the precision of the activations to be slightly larger than one bit. For instance, Quantized Neural Networks (QNN) [119], DoReFa-Net [120], and HWGQ-Net [130] allow the activations to have 2-bits, while the weights remain at 1-bit; in HWGQ-Net, this reduces the accuracy loss to 5.2%.

All the previously described binary nets limit the weights to two values (-w and w); however, there may be benefits for allowing weights to be zero (i.e., -w, 0, w). Although this requires an additional bit per weight compared to binary weights, the sparsity of the weights can be exploited to reduce computation and storage cost, which can potentially cancel out the cost of the additional bit. This is explored in Ternary Weight Nets (TWN) [131] and then extended in Trained Ternary Quantization (TTQ) where a different scale is trained for each weight (i.e., -w_1, 0, w_2) for an accuracy loss of 0.6% [132], assuming 32-bit floating point for the activations.

Hardware implementations for binary/ternary nets have been explored in recent publications. YodaNN [133] uses binary weights, while BRein [134] uses binary weights and activations. Binary weights are also used in the compute in SRAM work [102] described in Section VI. Finally, the nominally spike-inspired TrueNorth chip can implement a reduced precision neural network with binary activations and ternary weights using TrueNorth's quantized weight table [9]. These works tend not to support state-of-the-art DNN models (with the exception of YodaNN).

2) Non-linear quantization: The previous works described involve linear quantization where the levels are uniformly spaced out. It has been shown that the distributions of the weights and activations are not uniform [118, 135], and thus a non-linear quantization can potentially improve accuracy. Specifically, there have been two popular approaches taken in recent works: (1) log domain quantization; (2) learned quantization or weight sharing.

Log domain quantization If the quantization levels are assigned based on a logarithmic distribution as shown in Fig 37(b), the weights and activations are more equally distributed across the different levels and each level is used more efficiently resulting in less quantization error. For instance, using 4 bits in linear quantization results in a 27.8% loss in accuracy versus a 5% loss for log base-2 quantization for VGG-16 [117]. Furthermore, when weights are quantized to powers of two, the multiplication can be replaced with a bit-shift [122, 135].[10] Incremental Network Quantization (INQ) can be used to further reduce the loss in accuracy by dividing the large and small weights into different groups, and then iteratively quantizing and re-training the weights [136].

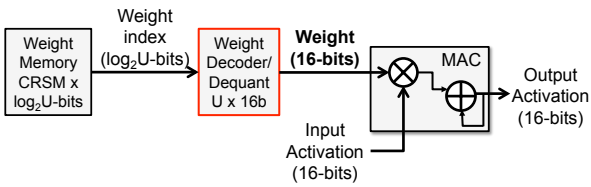


Fig. 40. Weight sharing hardware.

Weight Sharing forces several weights to share a single value. This reduces the number of unique weights in a filter or a layer. One example is to group the weights by using a hashing function and use one value for each group [137]. Alternatively, the weights can be grouped by the k-means algorithm [118]. Both the shared weights and the indexes indicating which weight to use at each position of the filter are stored. This leads to a two step process to fetch the weight: (1) read the weight index; (2) using the weight index, read the shared weights. This approach can reduce the cost of reading and storing the weights if the weight index (log_2 of the number of unique weights) is less than the bitwidth of the weight itself.

For instance, in Deep Compression [118], the number of unique weights per layer is reduced to 256 for convolutional layers and 16 for fully-connected layers in AlexNet, requiring 8-bit and 4-bit weight indexes, respectively. Assuming there are U unique weights and the size of the filters in the layer is C×R×S×M from Fig. 9(b), there will be energy savings if reading from a $CRSM \times log_2U$-bit memory plus a U×16-bit memory (as shown in Fig. 40) cost less than reading from a $CRSM$×16-bit memory. Note that unlike the previous quantization methods, the weight sharing approach does not reduce the precision of the MAC computation itself and only reduces the weight storage requirement.

B. Reduce Number of Operations and Model Size

In addition to reducing the size of each operation or operand (weight/activation), there is also a significant amount of research on methods to reduce the number of operations and model size. These techniques can be loosely classified as exploiting activation statistics, network pruning, network architecture design and knowledge distillation.

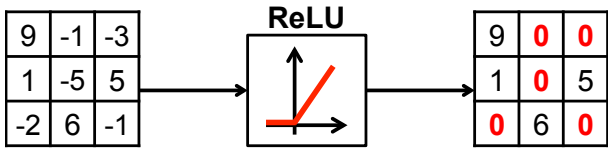
1) Exploiting Activation Statistics: As discussed in Section III-A1, ReLU is a popular form of non-linearity used in DNNs that sets all negative values to zero as shown in Fig. 41(a). As a result, the output activations of the feature maps after the ReLU are sparse; for instance, the feature maps in AlexNet have sparsity between 19% to 63% as shown in Fig. 41(b). This sparsity gives ReLU an implementation advantage over other non-linearities such as sigmoid, etc.

[9]This can also be thought of as a form of weights sharing, where only two weights are used per filter.

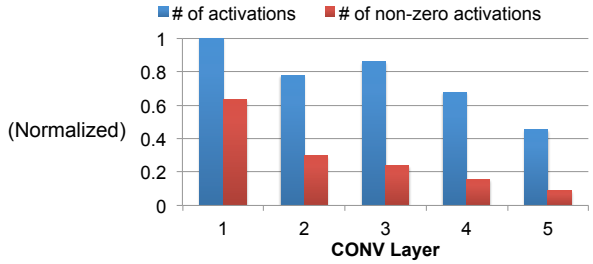
[10]Note however that multiplications do not account for a significant portion of the total energy.

Reduce Precision Method		bitwidth		Accuracy loss vs. 32-bit float (%)
		Weights	Activations	
Dynamic Fixed Point	w/o fine-tuning [121]	8	10	0.4
	w/ fine-tuning [122]	8	8	0.6
Reduce Weight	BinaryConnect [127]	1	32 (float)	19.2
	Binary Weight Network (BWN) [129]	1*	32 (float)	0.8
	Ternary Weight Networks (TWN) [131]	2*	32 (float)	3.7
	Trained Ternary Quantization (TTQ) [132]	2*	32 (float)	0.6
Reduce Weight and Activation	XNOR-Net [129]	1*	1*	11
	Binarized Neural Networks (BNN) [128]	1	1	29.8
	DoReFa-Net [120]	1*	2*	7.63
	Quantized Neural Networks (QNN) [119]	1	2*	6.5
	HWGQ-Net [130]	1*	2*	5.2
Non-linear Quantization	LogNet [135]	5 (conv), 4 (fc)	4	3.2
	Incremental Network Quantization (INQ) [136]	5	32 (float)	-0.2
	Deep Compression [118]	8 (conv), 4 (fc)	16	0
		4 (conv), 2 (fc)	16	2.6

TABLE III
METHODS TO REDUCE NUMERICAL PRECISION FOR ALEXNET. ACCURACY MEASURED FOR TOP-5 ERROR ON IMAGENET. *NOT APPLIED TO FIRST AND/OR LAST LAYERS



(a) ReLU non-linearity



(b) Distribution of activation after ReLU of AlexNet

Fig. 41. Sparsity in activations due to ReLU.

The sparsity can be exploited for energy and area savings using compression, particularly for off-chip DRAM access which is expensive. For instance, a simple run length coding that involves signaling non-zero values of 16-bits and then runs of zeros up to 31 can reduce the external memory bandwidth of the activations by 2.1× and the overall external bandwidth (including weights) by 1.5× [61].[11] In addition to compression, the hardware can also be modified such that it skips reading the weights and performing the MAC for zero-valued activations to reduce energy cost by 45% [94]. Rather than just gating the read and MAC computation, the hardware could also skip the cycle to increase the throughput by 1.37× [138].

The activations can be made to be even more sparse by pruning the low-valued activations. For instance, if all activations with small values are pruned, this can be translated into an additional 11% speed up [138] or 2× power reduction [139] with little impact on accuracy. Aggressively pruning more activations can provide additional throughput improvement at a cost of reduced accuracy.

2) Network Pruning: To make network training easier, the networks are usually over-parameterized. Therefore, a large amount of the weights in a network are redundant and can be removed (i.e., set to zero). This process is called network pruning. Aggressive network pruning often requires some fine-tuning of the weights to maintain the original accuracy. This was first proposed in 1989 through a technique called Optimal Brain Damage [140]. The idea was to compute the impact of each weight on the training loss (discussed in Section II-C), referred to as the weight saliency. The low-saliency weights were removed and the remaining weights were fine-tuned; this process was repeated until the desired weight reduction and accuracy were reached.

In 2015, a similar idea was applied to modern DNNs in [141]. Rather than using the saliency as a metric, which is too difficult to compute for the large-scaled DNNs, the pruning was simply based on the magnitude of the weights. Small weights were pruned and the model was fine-tuned to restore the accuracy. Without fine-tuning the weights, about 50% of the weights could be pruned. With fine-tuning, over 80% of the weights were pruned. Overall this approach can reduce the number of weights in AlexNet by 9× and the number of MACs by 3×. Most of the weight reduction comes from the fully-connected layers (9.9× for fully-connected layers versus 2.7× for convolutional layers).

However, the number of weights alone is not a good metric for energy. For instance, in AlexNet, the number of weights in the fully-connected layers is much larger than in the convolutional layers; however, the energy of the convolutional layers is much higher than the fully-connected layers as shown in Fig. 35 [80]. Rather than using the number of weights and MAC operations as proxies for energy, the pruning of the weights can be directly driven by energy itself [142]. An energy evaluation method can be used to estimate the DNN energy that accounts for the data movement from different levels of the memory hierarchy, the number of MACs, and the data sparsity as shown in Fig. 42; this energy estimation tool is available at [143]. The resulting energy values for popular DNN models are shown in Fig. 43(a). Energy-aware pruning

[11] This simple run length compression is within 5-10% of the theoretical entropy limit.

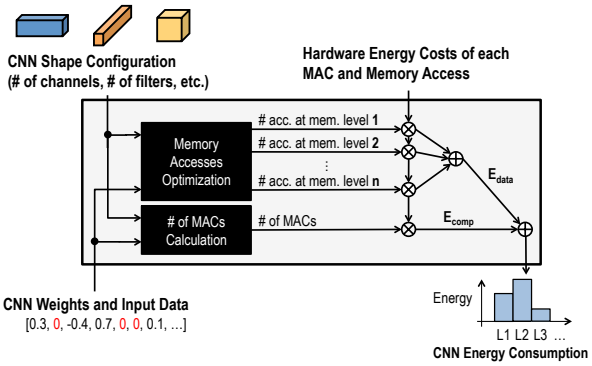
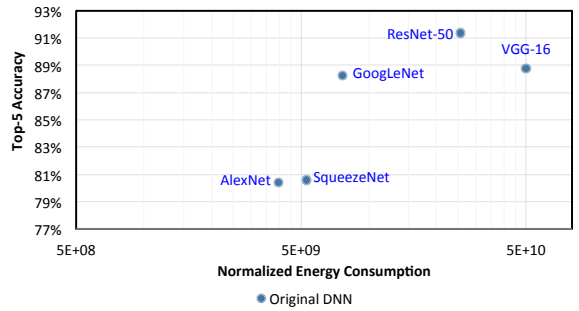
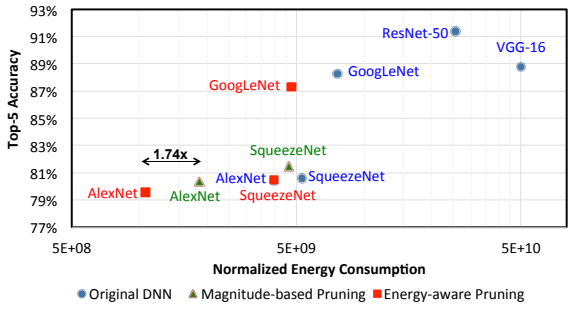


Fig. 42. Energy estimation methodology from [142], which estimates the energy based on data movement from different levels of the memory hierarchy, number of MACs, and data sparsity.



(a) Energy versus accuracy trade-off of popular DNN models.



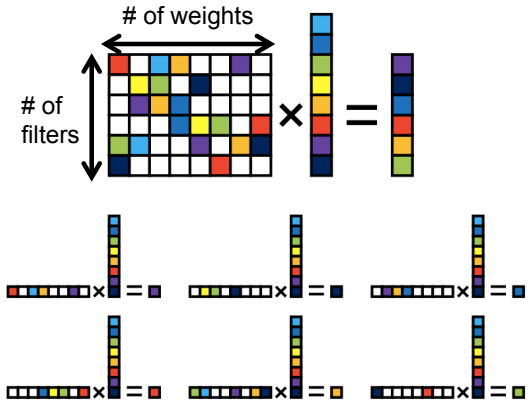
(b) Impact of energy-aware pruning.

Fig. 43. Energy values estimated with methodology in [142].

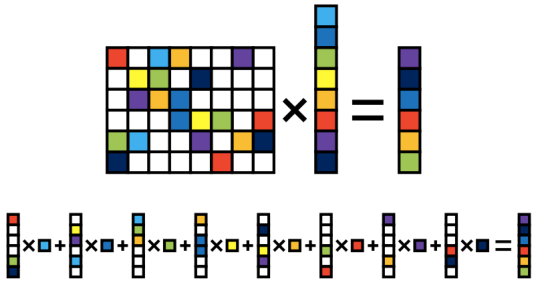
can then be used to prune weights based on energy to reduce the overall energy across all layers by 3.7× for AlexNet, which is 1.74× more efficient than magnitude-based approaches [141] as shown in Fig. 43(b). As mentioned previously, it is well known that AlexNet is over-parameterized. The energy-aware pruning can also be applied to GoogleNet, which is already a small DNN model, for a 1.6× energy reduction.

Recent works have examine how to efficiently support processing of sparse weights in hardware. One area of interest is how to best store the sparse weights after pruning. Similar to compressing the sparse activations discussed in Section VII-B1, the sparse weights can be compressed to reduce memory access bandwidth by 20 to 30% [118].

When DNN processing is performed as a matrix-vector



(a) Compressed sparse row (CSR)



(b) Compressed sparse column (CSC)

Fig. 44. Sparse matrix-vector multiplications using different storage formats (Figure from [144]).

multiplication, as shown in Fig. 18(a), one challenge is to determine how to store the sparse weight matrix in a compressed format. The compression can be applied either in row or column order. A compressed sparse row (CSR) format, as shown in Fig. 44(a), is often used to perform Sparse Matrix-Vector multiplication. However, the input vector needs to be read in multiple times even though only a subset of it is used since each row of the matrix is sparse. Alternatively, a compressed sparse column (CSC) format, as shown in Fig. 44(b), can be used, where the output is updated several times, and only one element of the input vector is read at a time [144]. The CSC format will provide an overall lower memory bandwidth than CSR if the output is smaller than the input, or in the case of DNN, if the number of filters is *not* significantly larger than the number of weights in the filter ($C \times R \times S$ from Fig. 9(b)). Since this is often true, CSC can be an effective format for sparse DNN processing.

Custom hardware has been explored to efficiently support pruned DNN models. Many works aim to perform the processing without decompressing the weights or activations. EIE [145] performs the sparse matrix-vector multiplication specifically for the fully connected layers. It stores the weights in a CSC format along with the start location of each column, which needs to be stored since the compressed weights have variable length. When the input is not zero, the compressed weight column is read and the output is updated. To handle the sparsity, additional logic is used to keep track of the location of the output that should be updated. SCNN [146] supports processing of convolutional

layers in a compressed format. It uses an input stationary dataflow to deliver the compressed weights and activations to a multiplier array followed by a scatter network to add the scattered partial sums.

Recent works have also explored the use of structured pruning to avoid the need for custom hardware [147, 148]. Rather than pruning individual weights (also referred to as fine-grained pruning), structured pruning involves pruning groups of weights (also referred to as coarse-grained pruning). The benefits of structured pruning are (1) the resulting weights can better align with the data-parallel architecture (e.g., SIMD) found in existing general purpose hardware, which results in more efficient processing [149]; (2) it amortizes the overhead cost required to signal the location of the non-zero weights across a group of weights, which improves compression and thus reduces storage cost. These groups of weights can include a pair of neighboring weights, an entire row or column of a filter, an entire channel of a filter or the entire filter itself; using larger groups tends to result in higher loss in accuracy [150].

3) Compact Network Architectures: The number of weights and operations can also be reduced by improving the network architecture itself. The trend is to replace a large filter with a series of smaller filters, which have fewer weights in total; when the filters are applied sequentially, they achieve the same overall effective receptive field (i.e., the region the filter uses from input image to compute an output). This approach can be applied during the network architecture design (before training) or by decomposing the filters of a trained network (after training). The latter one avoids the hassle of training networks from scratch. However, it is less flexible than the former one. For example, existing methods can only decompose a filter in a trained network into a series of filters without non-linearity between them.

a) Before Training: In recent DNN models, filters with a smaller width and height are used more frequently because concatenating several of them can emulate a larger filter as shown in Fig. 13. For example, one 5×5 convolution can be replaced with two 3×3 convolutions. Alternatively, one N×N convolution can be decomposed into two 1-D convolutions, one 1×N and one N×1 convolution [53]; this basically imposes a restriction that the 2-D filter must be separable, which is a common constraint in image processing [151]. Similarly, a 3-D convolution can be replaced by a set of 2-D convolutions (i.e., applied only on one of the input channels) followed by 1×1 3-D convolutions as demonstrated in Xception [152] and MobileNets [153]. The order of the 2-D convolutions and 1×1 3-D convolutions can be switched.

1×1 convolutional layers can also be used to reduce the number of channels in the output feature map for a given layer, which reduces the number of filter channels and thus computation cost for the filters in the next layer as demonstrated in [15, 51, 52]; this is often referred to as a 'bottleneck' as discussed in Section III-B. For this purpose, the number of 1×1 filters has to be less than the number of channels in the 1×1 filter. For example, 32 filters of 1×1×64 can transform an input with 64 channels to an output of 32 channels and reduce the number of filter channels in the next layer to 32. SqueezeNet uses many 1×1 filters to aggressively reduce the number of weights [154]. It proposes a *fire* module that first 'squeezes' the network with 1×1 convolution filters and then expands it with multiple 1×1 and 3×3 convolution filters. It achieves an overall 50× reduction in number of weights compared to AlexNet, while maintaining the same accuracy. It should be noted, however, that reducing the number of weights does not necessarily reduce energy; for instance, SqueezeNet consumes more energy than AlexNet, as shown in Fig. 43(a).

b) After Training: Tensor decomposition can be used to decompose filters in a trained network without impacting the accuracy. It treats weights in a layer as a 4-D tensor and breaks it into a combination of smaller tensors (i.e., several layers). Low-rank approximation can then be applied to further increase the compression rate at the cost of accuracy degradation, which can be restored by fine-tuning the weights.

This approach is demonstrated using Canonical Polyadic (CP) decomposition, a high-order extension of singular value decomposition that can be solved by various methods, such as a greedy algorithm [155] or a non-linear least-square method [156]. Combining CP-decomposition with low-rank approximation achieves a 4.5× speed-up on CPUs [156]. However, CP-decomposition cannot be computed in a numerically stable way when the dimension of the tensor, which represents the weights, is larger than two [156]. To alleviate this problem, Tucker decomposition is adopted instead in [157].

4) Knowledge Distillation: Using a deep network or averaging the predictions of different models (i.e., ensemble) gives a better accuracy than using a single shallower network. However, the computational complexity is also higher. To get the best of both worlds, knowledge distillation transfers the knowledge learned by the complex model (teacher) to the simpler model (student). The student network can therefore achieve an accuracy that would be unachievable if it was directly trained with the same dataset [158, 159]. For example, [160] shows how using knowledge distillation can improve the speech recognition accuracy of a student net by 2%, which is similar to the accuracy of a teacher net that is composed of an ensemble of 10 networks.

Fig. 45 shows the simplest knowledge distillation method [158]. The softmax layer is commonly used as the output layer in the image classification networks to generate the class probabilities from the class scores[12]; it squashes the class scores into values between 0 and 1 that sum up to 1. For this knowledge distillation method, soft targets (values between 0 and 1) such as the class scores of the teacher DNN (or an ensemble of teacher DNNs) are used instead of the hard targets (values of either 0 or 1) such as the labels in the dataset. The objective is to minimize the squared difference between the soft targets and the class scores of the student DNN. Class scores are used as the soft targets instead of the class probabilities because small values in the class scores contain important information that may be eliminated by the softmax. Alternatively, class probabilities after the softmax layer can be used as soft targets if the softmax is configured to generate softer class probabilities where the smaller values retain more information [160]. Finally, the intermediate representations of

[12]Also commonly referred to as logits.

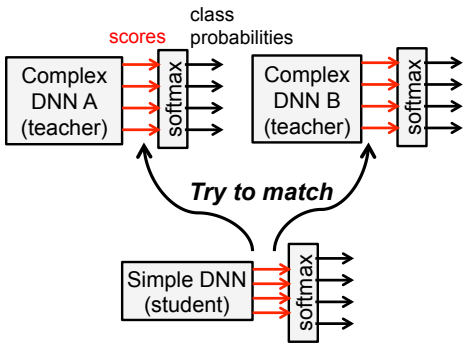


Fig. 45. Knowledge distillation matches the class scores of a small DNN to an ensemble of large DNNs.

the teacher DNN can also be incorporated as the extra hints to train the student DNN [161].

VIII. Benchmarking Metrics for DNN Evaluation and Comparison

As we have seen in this article, there has been a significant amount of research on efficient processing of DNNs. We should consider several key metrics to compare the various strengths and weaknesses of different designs and proposed techniques. These metrics should cover important attributes such as accuracy/robustness, power/energy consumption, throughput/latency and cost. Reporting all these metrics is important in order to provide a complete picture of the trade-offs made by a proposed design or technique. We have prepared a website to collect these metrics from various publications [162].

In terms of *accuracy* and *robustness*, it is important that the accuracy be reported on widely-accepted datasets as discussed in Section IV. The difficulty of the dataset and/or task should be considered when measuring the accuracy. For instance, the MNIST dataset for digit recognition is significantly easier than the ImageNet dataset. As a result, a DNN that performs well on MNIST may not necessarily perform well on ImageNet. Thus it is important that the same dataset and task is used when comparing the accuracy of different DNN models; currently ImageNet is preferred since it presents a challenge for DNNs, as opposed to MNIST, which can also be addressed with simple non-DNN techniques. To demonstrate primarily hardware innovations, it would be desirable to report results for widely-used DNN models (e.g., AlexNet, GoogLeNet) whose accuracy and robustness have been well studied and tested.

Energy and *power* are important when processing DNNs at the edge in embedded devices with limited battery capacity (e.g., smart phones, smart sensors, UAVs, and wearables), or in the cloud in data centers with stringent power ceilings due to cooling costs, respectively. Edge processing is preferred over the cloud for certain applications due to latency, privacy or communication bandwidth limitations. When evaluating the power and energy consumption, it is important to account for all aspects of the system including the chip and external memory accesses.

High throughput is necessary to deliver real-time performance for interactive applications such as navigation and robotics. For data analytics, high throughput means that more data can be analyzed in a given amount of time. As the amount of visual data is growing exponentially, high-throughput big data analytics becomes important, particularly if an action needs to be taken based on the analysis (e.g., security or terrorist prevention; medical diagnosis).

Low latency is necessary for real-time interactive applications. Latency measures the time between when the pixel arrives to a system and when the result is generated. Latency is measured in terms of seconds, while throughput is measured in operations/second. Often high throughput is obtained by batching multiple images/frames together for processing; this results in multiple frame latency (e.g., at 30 frames per second, a batch of 100 frames results in a 3 second delay). This delay is not acceptable for real-time applications, such as high-speed navigation where it would reduce the time available for course correction. Thus achieving low latency and high throughput simultaneously can be a challenge.

Hardware cost is in large part dictated by the amount of on-chip storage and the number of cores. Typical embedded processors have limited on-chip storage on the order of a few hundred kilobytes. Since there is a trade-off between the amount of on-chip memory and the external memory bandwidth, both metrics should be reported. Similarly, there is a correlation between the number of cores and the throughput. In addition, while many cores can be built on a chip, the number of cores that can actually be used at a given time should be reported. It is often unrealistic to assume peak utilization and performance due to limitations of mapping and memory bandwidth. Accordingly, the power and throughput should be reported for running actual DNNs as opposed to only reporting theoretical limits.

A. Metrics for DNN Models

To evaluate the properties of a given DNN model, we should consider the following metrics:

- The *accuracy* of the model in terms of the top-5 error on datasets such as ImageNet. Also, the type of data augmentation used (e.g., multiple crops, ensemble models) should be reported.
- The *network architecture* of the model should be reported, including number of layers, filter sizes, number of filters and number of channels.
- The *number of weights* impact the storage requirement of the model and should be reported. If possible, the number of non-zero weights should be reported since this reflects the theoretical minimum storage requirements.
- The *number of MACs* that needs to be performed should be reported as it is somewhat indicative of the number of operations and potential throughput of the given DNN. If possible, the number of non-zero MACs should also be reported since this reflects the theoretical minimum compute requirements.

Table IV shows how these metrics are reported for various well known DNNs. The accuracy is reported for the case where only a single crop for a single model is used for classification, such that the number of weights and MACs in the table are

Metrics	AlexNet		GoogLeNet v1	
	dense	sparse	dense	sparse
Top-5 error	19.6	20.4	11.7	12.7
Number of CONV Layers	5	5	57	57
Depth in (Number of CONV Layers)	5	5	21	21
Filter Sizes	3,5,11		1,3,5,7	
Number of Channels	3-256		3-832	
Number of Filters	96-384		16-384	
Stride	1,4		1,2	
NZ Weights	2.3M	351k	6.0M	1.5M
NZ MACs	395M	56.4M	806M	220M
FC Layers	3	3	1	1
Filter Sizes	1,6		1	
Number of Channels	256-4096		1024	
Number of Filters	1000-4096		1000	
NZ Weights	58.6M	5.4M	1M	870k
NZ MACs	14.5M	1.9M	635k	663k
Total NZ Weights	61M	5.7M	7M	2.4M
Total NZ MACs	410M	58.3M	806M	221M

TABLE IV
METRICS FOR POPULAR DNN MODELS. SPARSITY IS ACCOUNT FOR BY REPORTING NON-ZERO (NZ) WEIGHTS AND MACS.

consistent.[13] Note that accounting for the number of non-zero (NZ) operations significantly reduces the number of MACs and weights. Since the number of NZ MACs depends on the input data, we propose using the publicly available 50,000 validation images from ImageNet for the computation. Finally, there are various methods to reduce the weights in a DNN (e.g., network pruning in Section VII-B2). Table IV shows another example of these DNN model metrics, by comparing sparse DNNs pruned using [142] to dense DNNs.

B. Metrics for DNN Hardware

To measure the efficiency of the DNN hardware, we should consider the following additional metrics:

- The *power and energy* consumption of the design should be reported for various DNN models; the DNN model specifications should be provided including which layers and bit precision are supported by the hardware during measurement. In addition, the amount of off-chip accesses (e.g., DRAM accesses) should be included since it accounts for a significant portion of the system power; it can be reported in terms of the total amount of data that is read and written off-chip per inference.
- The *latency and throughput* should be reported in terms of the batch size and the actual run time for various DNN models, which accounts for mapping and memory bandwidth effects. This provides a more useful and informative metric than peak throughput.
- The *cost* of the chip depends on the area efficiency, which accounts for the size and type of memory (e.g., registers or SRAM) and the amount of control logic. It should be reported in terms of the core area in squared millimeters per multiplier along with process technology.

In terms of cost, different platforms will have different implementation-specific metrics. For instance, for an FPGA, the specific device should be reported, along with the utilization of resources such as DSP, BRAM, LUT and FF; performance density such as GOPs/slice can also be reported.

Each processor should report various specifications for each metric as shown in Table V, using the Eyeriss chip as an example. It is important that all metrics and specifications are accounted for in order fairly evaluate all the design trade-offs. For instance, without the accuracy given for a specific dataset and task, one could run a simple DNN and easily claim low power, high throughput, and low cost – however, the processor might not be usable for a meaningful task; alternatively, without reporting the off-chip bandwidth, one could build a processor with only multipliers and easily claim low cost, high throughput, high accuracy, and low *chip* power – however, when evaluating *system* power, the off-chip memory access would be substantial. Finally, the test setup should also be reported, including whether the results are measured or obtained from simulation[14] and how many images were tested.

In summary, the evaluation process for whether a DNN system is a viable solution for a given application might go as follows: (1) the accuracy determines if it can perform the given task; (2) the latency and throughput determine if it can run fast enough and in real-time; (3) the energy and power consumption will primarily dictate the form factor of the device where the processing can operate; (4) the cost, which is primarily dictated by the chip area, determines how much one would pay for this solution.

IX. SUMMARY

The use of deep neural networks (DNNs) has seen explosive growth in the past few years. They are currently widely used for many artificial intelligence (AI) applications including computer vision, speech recognition and robotics and are often delivering better than human accuracy. However, while DNNs can deliver this outstanding accuracy, it comes at the cost of high computational complexity. Consequently, techniques that enable efficient processing of deep neural network to improve *energy-efficiency* and *throughput* without sacrificing *accuracy* with cost-effective hardware are critical to expanding the deployment of DNNs in both existing and new domains.

Creating a system for efficient DNN processing should begin with understanding the current and future applications and the specific computations required both now and the potential evolution of those computations. This article surveys a number of the current applications, focusing on computer vision applications, the associated algorithms, and the data being used to drive the algorithms. These applications, algorithms and input data are experiencing rapid change. So extrapolating these trends to determine the degree of flexibility desired to handle next generation computations, becomes an important ingredient of any design project.

[13]Data augmentation is often used to increase accuracy. This includes using multiple crops of an image to account for misalignment; in addition, an ensemble of multiple models can be used where each model has different weights due to different training settings, such as using different initializations or datasets, or even different network architectures. If multiple crops and models are used, then the number of MACs and weights required would increase.

[14]If obtained from simulation, it should be clarified whether it is from synthesis or post place-and-route and what library corner was used.

Metrics	Specifications	Eyeriss	
Cost	**Process Technology**	65nm LP TSMC (1.0V)	
	Total Core Area (mm^2)	12.25	
	Total On-chip Memory (kB)	192	
	Number of Multipliers	168	
	Core area per Multiplier (mm^2)	0.073	
	On-chip Memory per Multiplier (kB)	1.14	
Test Setup	**Measured or Simulated**	Measured	
	If Simulated, Syn or PnR	n/a	
Accuracy	**DNN Model**	AlexNet	VGG-16
	Top-5 error on ImageNet	19.8	8.8
	Dense/Sparse	Dense	Dense
	Supported Layers	All CONV layers	All CONV layers
	Bits per Weight	16	16
	Bits per Input Activation	16	16
Latency and Throughput	**Batch Size**	4	3
	Run Time (msec)	115.3	4309.4
Power and Energy	**Power (mW)**	278	236
	Off-chip Accesses per Image Inference (MBytes)	3.85	107.03
Test Setup	**Number of Images Tested**	100	100

TABLE V
EXAMPLE BENCHMARK METRICS FOR EYERISS [94].

During the design-space exploration process, it is critical to understand and balance the important system metrics. For DNN computation these include the accuracy, energy, throughput and hardware cost. Evaluating these metrics is, of course, key, so this article surveys the important components of a DNN workload. In specific, a DNN workload has two major components. First, the workload is the form of each DNN network including the 'shape' of each layer and the interconnections between layers. These can vary both within and between applications. Second, the workload consists of the specific the data input to the DNN. This data will vary with the input set used for training or the data input during operation for inference.

This article also surveys a number of avenues that prior work have taken to optimize DNN processing. Since data movement dominates energy consumption, a primary focus of some recent research has been to reduce data movement while maintaining accuracy, throughput and cost. This means selecting architectures with favorable memory hierarchies like a spatial array, and developing dataflows that increase data reuse at the low-cost levels of the memory hierarchy. We have included a taxonomy of dataflows and an analysis of their characteristics. Other work is presented that aims to save space and energy by changing the representation of data values in the DNN. Still other work saves energy and sometimes increases throughput by exploiting the sparsity of weights and/or activations.

The DNN domain also affords an excellent opportunity for joint hardware/software co-design. For example, various efforts have noted that efficiency can be improved by increasing sparsity (increasing the number of zero values) or optimizing the representation of data by reducing the precision of values or using more complex mappings of the stored value to the actual value used for computation. However, to avoid losing accuracy it is often useful to modify the network or fine-tune the network's weights to accommodate these changes. Thus, this article both reviews a variety of these techniques and discusses the frameworks that are available for describing, running and training networks.

Finally, DNNs afford the opportunity to use mixed-signal circuit design and advanced technologies to improve efficiency. These include using memristors for analog computation and 3-D stacked memory. Advanced technologies can also can facilitate moving computation closer to the source by embedding computation near or within the sensor and the memories. Of course, all of these techniques should also be considered in combination, while being careful to understand their interactions and looking for opportunities for joint hardware/algorithm co-optimization.

In conclusion, although much work has been done, deep neural networks remain an important area of research with many promising applications and opportunities for innovation at various levels of hardware design.

ACKNOWLEDGMENTS

Funding provided by DARPA YFA, MIT CICS, and gifts from Nvidia and Intel. The authors thank the anonymous reviewers as well as James Noraky, Mehul Tikekar and Zhengdong Zhang for providing valuable feedback on this paper.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015.
[2] L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams *et al.*, "Recent advances in deep learning for speech research at Microsoft," in *ICASSP*, 2013.
[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *NIPS*, 2012.
[4] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *ICCV*, 2015.
[5] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, "Dermatologist-level classification of skin

cancer with deep neural networks," *Nature*, vol. 542, no. 7639, pp. 115–118, 2017.

[6] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[7] F.-F. Li, A. Karpathy, and J. Johnson, "Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition," http://cs231n.stanford.edu/.

[8] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[9] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch *et al.*, "Convolutional networks for fast, energy-efficient neuromorphic computing," *Proceedings of the National Academy of Sciences*, 2016.

[10] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through FFTs," in *ICLR*, 2014.

[11] Y. LeCun, L. D. Jackel, B. Boser, J. S. Denker, H. P. Graf, I. Guyon, D. Henderson, R. E. Howard, and W. Hubbard, "Handwritten digit recognition: applications of neural network chips and automatic learning," *IEEE Commun. Mag.*, vol. 27, no. 11, pp. 41–46, Nov 1989.

[12] B. Widrow and M. E. Hoff, "Adaptive switching circuits," in *1960 IRE WESCON Convention Record*, 1960.

[13] B. Widrow, "Thinking about thinking: the discovery of the LMS algorithm," *IEEE Signal Process. Mag.*, 2005.

[14] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *CVPR*, 2016.

[16] "Complete Visual Networking Index (VNI) Forecast," Cisco, June 2016.

[17] J. Woodhouse, "Big, big, big data: higher and higher resolution video surveillance," technology.ihs.com, January 2016.

[18] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation," in *CVPR*, 2014.

[19] J. Long, E. Shelhamer, and T. Darrell, "Fully Convolutional Networks for Semantic Segmentation," in *CVPR*, 2015.

[20] K. Simonyan and A. Zisserman, "Two-stream convolutional networks for action recognition in videos," in *NIPS*, 2014.

[21] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, 2012.

[22] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *Journal of Machine Learning Research*, vol. 12, no. Aug, pp. 2493–2537, 2011.

[23] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *CoRR abs/1609.03499*, 2016.

[24] H. Y. Xiong, B. Alipanahi, L. J. Lee, H. Bretschneider, D. Merico, R. K. Yuen, Y. Hua, S. Gueroussov, H. S. Najafabadi, T. R. Hughes *et al.*, "The human splicing code reveals new insights into the genetic determinants of disease," *Science*, vol. 347, no. 6218, p. 1254806, 2015.

[25] J. Zhou and O. G. Troyanskaya, "Predicting effects of noncoding variants with deep learning-based sequence model," *Nature methods*, vol. 12, no. 10, pp. 931–934, 2015.

[26] B. Alipanahi, A. Delong, M. T. Weirauch, and B. J. Frey, "Predicting the sequence specificities of dna-and rna-binding proteins by deep learning," *Nature biotechnology*, vol. 33, no. 8, pp. 831–838, 2015.

[27] H. Zeng, M. D. Edwards, G. Liu, and D. K. Gifford, "Convolutional neural network architectures for predicting dna–protein binding," *Bioinformatics*, vol. 32, no. 12, pp. i121–i127, 2016.

[28] M. Jermyn, J. Desroches, J. Mercier, M.-A. Tremblay, K. St-Arnaud, M.-C. Guiot, K. Petrecca, and F. Leblond, "Neural networks improve brain cancer detection with raman spectroscopy in the presence of operating room light artifacts," *Journal of Biomedical Optics*, vol. 21, no. 9, pp. 094 002–094 002, 2016.

[29] D. Wang, A. Khosla, R. Gargeya, H. Irshad, and A. H. Beck, "Deep learning for identifying metastatic breast cancer," *arXiv preprint arXiv:1606.05718*, 2016.

[30] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[31] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," in *NIPS Deep Learning Workshop*, 2013.

[32] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *Journal of Machine Learning Research*, vol. 17, no. 39, pp. 1–40, 2016.

[33] M. Pfeiffer, M. Schaeuble, J. Nieto, R. Siegwart, and C. Cadena, "From Perception to Decision: A Data-driven Approach to End-to-end Motion Planning for Autonomous Ground Robots," in *ICRA*, 2017.

[34] S. Gupta, J. Davidson, S. Levine, R. Sukthankar, and J. Malik, "Cognitive mapping and planning for visual navigation," in *CVPR*, 2017.

[35] T. Zhang, G. Kahn, S. Levine, and P. Abbeel, "Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search," in *ICRA*, 2016.

[36] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Safe, multi-agent, reinforcement learning for autonomous driving," in *NIPS Workshop on Learning, Inference and Control of Multi-Agent Systems*, 2016.

[37] N. Hemsoth, "The Next Wave of Deep Learning Applications," Next Platform, September 2016.

[38] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[39] T. N. Sainath, A.-r. Mohamed, B. Kingsbury, and B. Ramabhadran, "Deep convolutional neural networks for LVCSR," in *ICASSP*, 2013.

[40] V. Nair and G. E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," in *ICML*, 2010.

[41] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *ICML*, 2013.

[42] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *ICCV*, 2015.

[43] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)," *ICLR*, 2016.

[44] X. Zhang, J. Trmal, D. Povey, and S. Khudanpur, "Improving deep neural network acoustic models using generalized maxout networks," in *ICASSP*, 2014.

[45] Y. Zhang, M. Pezeshki, P. Brakel, S. Zhang, , C. Laurent, Y. Bengio, and A. Courville, "Towards End-to-End Speech Recognition with Deep Convolutional Neural Networks," in *Interspeech*, 2016.

[46] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional

architecture for fast feature embedding," in *ACM International Conference on Multimedia*, 2014.
[47] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *ICML*, 2015.
[48] 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.
[49] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks," in *ICLR*, 2014.
[50] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *ICLR*, 2015.
[51] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper With Convolutions," in *CVPR*, 2015.
[52] M. Lin, Q. Chen, and S. Yan, "Network in Network," in *ICLR*, 2014.
[53] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *CVPR*, 2016.
[54] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," in *AAAI*, 2017.
[55] G. Urban, K. J. Geras, S. E. Kahou, O. Aslan, S. Wang, R. Caruana, A. Mohamed, M. Philipose, and M. Richardson, "Do Deep Convolutional Nets Really Need to be Deep and Convolutional?" *ICLR*, 2017.
[56] "Caffe LeNet MNIST," http://caffe.berkeleyvision.org/gathered/examples/mnist.html.
[57] "Caffe Model Zoo," http://caffe.berkeleyvision.org/model_zoo.html.
[58] "Matconvnet Pretrained Models," http://www.vlfeat.org/matconvnet/pretrained/.
[59] "TensorFlow-Slim image classification library," https://github.com/tensorflow/models/tree/master/slim.
[60] "Deep Learning Frameworks," https://developer.nvidia.com/deep-learning-frameworks.
[61] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE J. Solid-State Circuits*, vol. 51, no. 1, 2017.
[62] C. J. B. Yann LeCun, Corinna Cortes, "THE MNIST DATABASE of handwritten digits," http://yann.lecun.com/exdb/mnist/.
[63] L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *ICML*, 2013.
[64] A. Krizhevsky, V. Nair, and G. Hinton, "The CIFAR-10 dataset," https://www.cs.toronto.edu/~kriz/cifar.html.
[65] A. Torralba, R. Fergus, and W. T. Freeman, "80 million tiny images: A large data set for nonparametric object and scene recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, no. 11, pp. 1958–1970, 2008.
[66] A. Krizhevsky and G. Hinton, "Convolutional deep belief networks on cifar-10," *Unpublished manuscript*, vol. 40, 2010.
[67] B. Graham, "Fractional max-pooling," *arXiv preprint arXiv:1412.6071*, 2014.
[68] "Pascal VOC data sets," http://host.robots.ox.ac.uk/pascal/VOC/.
[69] "Microsoft Common Objects in Context (COCO) dataset," http://mscoco.org/.
[70] "Google Open Images," https://github.com/openimages/dataset.
[71] "YouTube-8M," https://research.google.com/youtube8m/.
[72] "AudioSet," https://research.google.com/audioset/index.html.
[73] S. Condon, "Facebook unveils Big Basin, new server geared for deep learning," ZDNet, March 2017.
[74] C. Dubout and F. Fleuret, "Exact acceleration of linear object detectors," in *ECCV*, 2012.
[75] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *ICANN*, 2014.
[76] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *CVPR*, 2016.
[77] "Intel Math Kernel Library," https://software.intel.com/en-us/mkl.
[78] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," *arXiv preprint arXiv:1410.0759*, 2014.
[79] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *ISSCC*, 2014.
[80] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *ISCA*, 2016.
[81] ——, "Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators," *IEEE Micro's Top Picks from the Computer Architecture Conferences*, vol. 37, no. 3, May-June 2017.
[82] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A Massively Parallel Coprocessor for Convolutional Neural Networks," in *ASAP*, 2009.
[83] V. Sriram, D. Cox, K. H. Tsoi, and W. Luk, "Towards an embedded biologically-inspired machine vision processor," in *FPT*, 2010.
[84] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A Dynamically Configurable Coprocessor for Convolutional Neural Networks," in *ISCA*, 2010.
[85] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks," in *CVPR Workshop*, 2014.
[86] S. Park, K. Bong, D. Shin, J. Lee, S. Choi, and H.-J. Yoo, "A 1.93TOPS/W scalable deep learning/inference processor with tetra-parallel MIMD architecture for big-data applications," in *ISSCC*, 2015.
[87] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, "Origami: A Convolutional Network Accelerator," in *GLVLSI*, 2015.
[88] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep Learning with Limited Numerical Precision," in *ICML*, 2015.
[89] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting Vision Processing Closer to the Sensor," in *ISCA*, 2015.
[90] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for Convolutional Neural Networks," in *ICCD*, 2013.
[91] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *FPGA*, 2015.
[92] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning," in *ASPLOS*, 2014.
[93] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A Machine-Learning Supercomputer," in *MICRO*, 2014.
[94] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *ISSCC*, 2016.
[95] V. Sze, M. Budagavi, and G. J. Sullivan, "High Efficiency Video Coding (HEVC): Algorithms and Architectures," in *Integrated Circuit and Systems*. Springer, 2014, pp. 1–375.
[96] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *MICRO*, 2016.
[97] D. Keitel-Schulz and N. Wehn, "Embedded DRAM development: Technology, physical design, and application issues," *IEEE Des. Test. Comput.*, vol. 18, no. 3, pp. 7–15, 2001.
[98] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM

architecture increases density and performance," in *Symp. on VLSI*, 2012.
[99] J. Standard, "High bandwidth memory (HBM) DRAM," *JESD235*, 2013.
[100] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory," in *ISCA*, 2016.
[101] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *ASPLOS*, 2017.
[102] J. Zhang, Z. Wang, and N. Verma, "A machine-learning classifier implemented in a standard 6T SRAM array," in *Symp. on VLSI*, 2016.
[103] Z. Wang, R. Schapire, and N. Verma, "Error-adaptive classifier boosting (EACB): Exploiting data-driven training for highly fault-tolerant hardware," in *ICASSP*, 2014.
[104] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," in *ISCA*, 2016.
[105] L. Chua, "Memristor-the missing circuit element," *IEEE Trans. Circuit Theory*, vol. 18, no. 5, pp. 507–519, 1971.
[106] L. Wilson, "International technology roadmap for semiconductors (ITRS)," *Semiconductor Industry Association*, 2013.
[107] Lu, Darsen, "Tutorial on Emerging Memory Devices," 2016.
[108] S. B. Eryilmaz, S. Joshi, E. Neftci, W. Wan, G. Cauwenberghs, and H.-S. P. Wong, "Neuromorphic architectures with electronic synapses," in *ISQED*, 2016.
[109] P. Chi, S. Li, Z. Qi, P. Gu, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A Novel Processing-In-Memory Architecture for Neural Network Computation in ReRAM-based Main Memory," in *ISCA*, 2016.
[110] M. Prezioso, F. Merrikh-Bayat, B. Hoskins, G. Adam, K. K. Likharev, and D. B. Strukov, "Training and operation of an integrated neuromorphic network based on metal-oxide memristors," *Nature*, vol. 521, no. 7550, pp. 61–64, 2015.
[111] J. Zhang, Z. Wang, and N. Verma, "A matrix-multiplying ADC implementing a machine-learning classifier directly with data conversion," in *ISSCC*, 2015.
[112] E. H. Lee and S. S. Wong, "A 2.5 GHz 7.7 TOPS/W switched-capacitor matrix multiplier with co-designed local memory in 40nm," in *ISSCC*, 2016.
[113] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, "RedEye: analog ConvNet image sensor architecture for continuous mobile vision," in *ISCA*, 2016.
[114] A. Wang, S. Sivaramakrishnan, and A. Molnar, "A 180nm CMOS image sensor with on-chip optoelectronic image compression," in *CICC*, 2012.
[115] H. Chen, S. Jayasuriya, J. Yang, J. Stephen, S. Sivaramakrishnan, A. Veeraraghavan, and A. Molnar, "ASP Vision: Optically Computing the First Layer of Convolutional Neural Networks using Angle Sensitive Pixels," in *CVPR*, 2016.
[116] A. Suleiman and V. Sze, "Energy-efficient HOG-based object detection at 1080HD 60 fps with multi-scale support," in *SiPS*, 2014.
[117] E. H. Lee, D. Miyashita, E. Chai, B. Murmann, and S. S. Wong, "Lognet: Energy-Efficient Neural Networks Using Logrithmic Computations," in *ICASSP*, 2017.
[118] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," in *ICLR*, 2016.
[119] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *arXiv preprint arXiv:1609.07061*, 2016.
[120] 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," *arXiv preprint arXiv:1606.06160*, 2016.
[121] Y. Ma, N. Suda, Y. Cao, J.-S. Seo, and S. Vrudhula, "Scalable and modularized RTL compilation of Convolutional Neural Networks onto FPGA," in *FPL*, 2016.
[122] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented Approximation of Convolutional Neural Networks," in *ICLR*, 2016.
[123] S. Higginbotham, "Google Takes Unconventional Route with Homegrown Machine Learning Chips," Next Platform, May 2016.
[124] T. P. Morgan, "Nvidia Pushes Deep Learning Inference With New Pascal GPUs," Next Platform, September 2016.
[125] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *MICRO*, 2016.
[126] B. Moons and M. Verhelst, "A 0.3–2.6 TOPS/W precision-scalable processor for real-time large-scale ConvNets," in *Symp. on VLSI*, 2016.
[127] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *NIPS*, 2015.
[128] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.
[129] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," in *ECCV*, 2016.
[130] Z. Cai, X. He, J. Sun, and N. Vasconcelos, "Deep learning with low precision by half-wave gaussian quantization," in *CVPR*, 2017.
[131] F. Li and B. Liu, "Ternary weight networks," in *NIPS Workshop on Efficient Methods for Deep Neural Networks*, 2016.
[132] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained Ternary Quantization," *ICLR*, 2017.
[133] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "YodaNN: An Ultra-Low Power Convolutional Neural Network Accelerator Based on Binary Weights," in *ISVLSI*, 2016.
[134] K. Ando, K. Ueyoshi, K. Orimo, H. Yonekawa, S. Sato, H. Nakahara, M. Ikebe, T. Asai, S. Takamaeda-Yamazaki, and M. Kuroda, T.and Motomura, "BRein Memory: A 13-Layer 4.2 K Neuron/0.8 M Synapse Binary/Ternary Reconfigurable In-Memory Deep Neural Network Accelerator in 65nm CMOS," in *Symp. on VLSI*, 2017.
[135] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional Neural Networks using Logarithmic Data Representation," *arXiv preprint arXiv:1603.01025*, 2016.
[136] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental Network Quantization: Towards Lossless CNNs with Low-precision Weights," in *ICLR*, 2017.
[137] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing Neural Networks with the Hashing Trick," in *ICML*, 2015.
[138] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: ineffectual-neuron-free deep neural network computing," in *ISCA*, 2016.
[139] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *ISCA*, 2016.
[140] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal Brain Damage," in *NIPS*, 1990.
[141] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *NIPS*, 2015.
[142] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning," in *CVPR*, 2017.
[143] "DNN Energy Estimation," http://eyeriss.mit.edu/energy.html.
[144] R. Dorrance, F. Ren, and D. Marković, "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs," in *ISFPGA*, 2014.
[145] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz,

and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," in *ISCA*, 2016.
[146] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *ISCA*, 2017.
[147] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *NIPS*, 2016.
[148] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *ACM Journal of Emerging Technologies in Computing Systems*, vol. 13, no. 3, p. 32, 2017.
[149] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *ISCA*, 2017.
[150] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the regularity of sparse structure in convolutional neural networks," in *CVPR Workshop on Tensor Methods In Computer Vision*, 2017.
[151] J. S. Lim, "Two-dimensional signal and image processing," *Englewood Cliffs, NJ, Prentice Hall, 1990, 710 p.*, 1990.
[152] F. Chollet, "Xception: Deep Learning With Depthwise Separable Convolutions," *CVPR*, 2017.
[153] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.
[154] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size," *ICLR*, 2017.
[155] E. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation," in *NIPS*, 2014.
[156] V. Lebedev, Y. Ganin, M. Rakhuba1, I. Oseledets, and V. Lempitsky, "Speeding-Up Convolutional Neural Networks Using Fine-tuned CP-Decomposition," *ICLR*, 2015.
[157] 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," in *ICLR*, 2016.
[158] C. Bucilu, R. Caruana, and A. Niculescu-Mizil, "Model Compression," in *SIGKDD*, 2006.
[159] L. Ba and R. Caurana, "Do Deep Nets Really Need to be Deep?" *NIPS*, 2014.
[160] G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network," in *NIPS Deep Learning Workshop*, 2014.
[161] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, "Fitnets: Hints for Thin Deep Nets," *ICLR*, 2015.
[162] "Benchmarking DNN Processors," http://eyeriss.mit.edu/benchmarking.html.