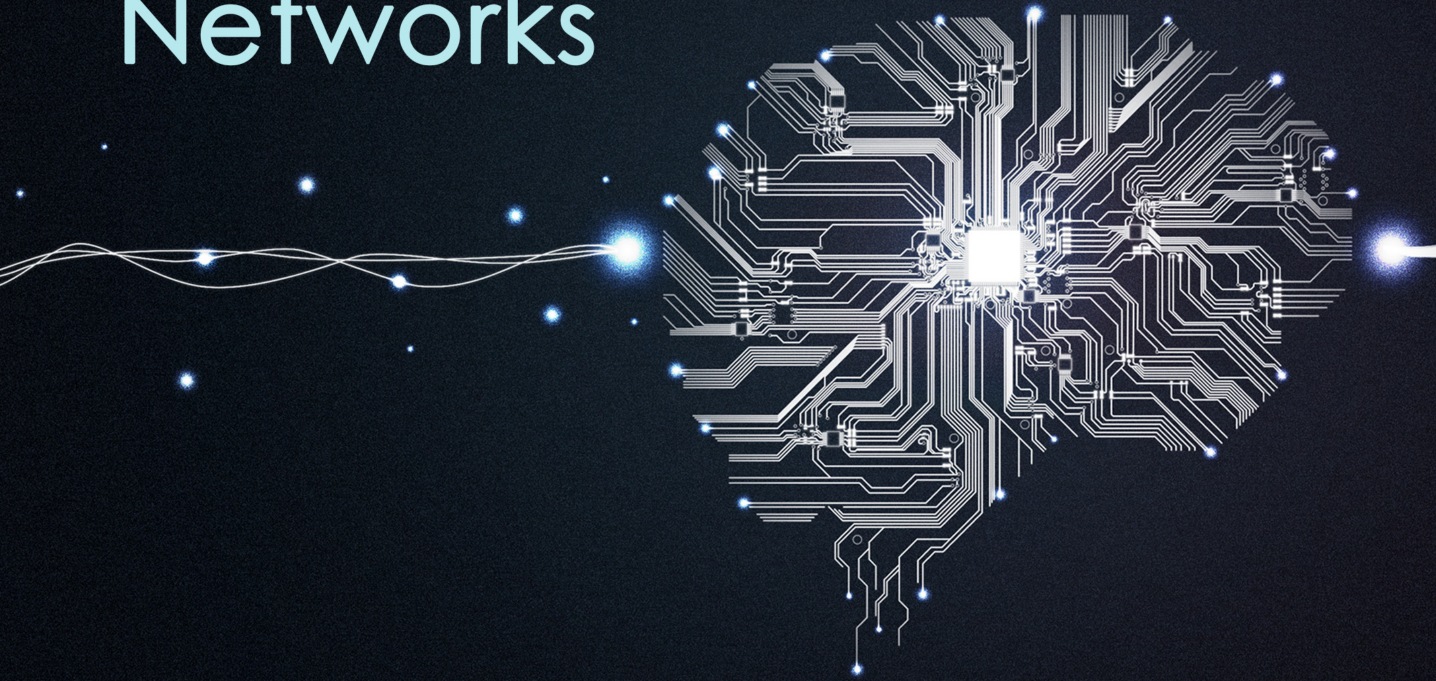
MORGAN & CLAYPOOL PUBLISHERS

Efficient Processing of Deep Neural Networks



Vivienne Sze, Yu-Hsin Chen,
Tien-Ju Yang, Joel Emer

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

Natalie Enright Jerger & Margaret Martonosi, *Series Editors*

Efficient Processing of Deep Neural Networks

Synthesis Lectures on Computer Architecture

Efficient Processing of Deep Neural Networks

Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer

www.morganclaypool.com

ISBN: 9781681738314 paperback
ISBN: 9781681738321 ebook
ISBN: 9781681738338 hardcover

DOI 10.2200/S01004ED1V01Y202004CAC050

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

Lecture #50
Series Editors: Natalie Enright Jerger, *University of Toronto*
Margaret Martonosi, *Princeton University*
Founding Editor Emeritus: Mark D. Hill, *University of Wisconsin, Madison*
Series ISSN
Print 1935-3235 Electronic 1935-3243

For book updates, sign up for mailing list at
http://mailman.mit.edu/mailman/listinfo/eems-news

Efficient Processing of Deep Neural Networks

Vivienne Sze, Yu-Hsin Chen, and Tien-Ju Yang
Massachusetts Institute of Technology

Joel S. Emer
Massachusetts Institute of Technology and Nvidia Research

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE #50

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

This book provides a structured treatment of the key principles and techniques for enabling efficient processing of deep neural networks (DNNs). 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. Therefore, techniques that enable efficient processing of deep neural networks to improve key metrics—such as energy-efficiency, throughput, and latency—without sacrificing accuracy or increasing hardware costs are critical to enabling the wide deployment of DNNs in AI systems.

The book includes background on DNN processing; a description and taxonomy of hardware architectural approaches for designing DNN accelerators; key metrics for evaluating and comparing different designs; features of DNN processing that are amenable to hardware/algorithm co-design to improve energy efficiency and throughput; and opportunities for applying new technologies. Readers will find a structured introduction to the field as well as formalization and organization of key concepts from contemporary work that provide insights that may spark new ideas.

KEYWORDS

deep learning, neural network, deep neural networks (DNN), convolutional neural networks (CNN), artificial intelligence (AI), efficient processing, accelerator architecture, hardware/software co-design, hardware/algorithm co-design, domain-specific accelerators

Contents

Preface

Deep neural networks (DNNs) have become extraordinarily popular; however, they come at the cost of high computational complexity. As a result, there has been tremendous interest in enabling efficient processing of DNNs. The challenge of DNN acceleration is threefold:

- to achieve high performance and efficiency,
- to provide sufficient flexibility to cater to a wide and rapidly changing range of workloads, and
- to integrate well into existing software frameworks.

In order to understand the current state of art in addressing this challenge, this book aims to provide an overview of DNNs, the various tools for understanding their behavior, and the techniques being explored to efficiently accelerate their computation. It aims to explain foundational concepts and highlight key design considerations when building hardware for processing DNNs rather than trying to cover all possible design configurations, as this is not feasible given the fast pace of the field (see Figure 1). It is targeted at researchers and practitioners who are familiar with computer architecture who are interested in how to efficiently process DNNs or how to design DNN models that can be efficiently processed. We hope that this book will provide a structured introduction to readers who are new to the field, while also formalizing and organizing key concepts to provide insights that may spark new ideas for those who are already in the field.

Organization

This book is organized into three modules that each consist of several chapters. The first module aims to provide an overall background to the field of DNN and insight on characteristics of the DNN workload.

- Chapter 1 provides background on the context of why DNNs are important, their history, and their applications.
- Chapter 2 gives an overview of the basic components of DNNs and popular DNN models currently in use. It also describes the various resources used for DNN research and development. This includes discussion of the various software frameworks and the public datasets that are used for training and evaluation.

The second module focuses on the design of hardware for processing DNNs. It discusses various architecture design decisions depending on the degree of customization (from general

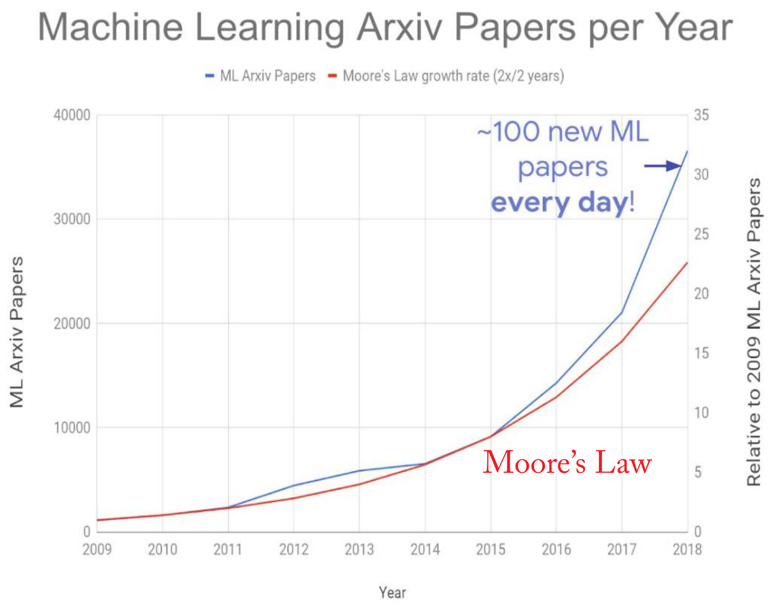


Figure 1: It's been observed that the number of ML publications are growing exponentially at a faster rate than Moore's law! (Figure from [1].)

purpose platforms to full custom hardware) and design considerations when mapping the DNN workloads onto these architectures. Both temporal and spatial architectures are considered.

- Chapter 3 describes the key metrics that should be considered when designing or comparing various DNN accelerators.

- Chapter 4 describes how DNN kernels can be processed, with a focus on temporal architectures such as CPUs and GPUs. To achieve greater efficiency, such architectures generally have a cache hierarchy and coarser-grained computational capabilities, e.g., vector instructions, making the resulting computation more efficient. Frequently for such architectures, DNN processing can be transformed into a matrix multiplication, which has many optimization opportunities. This chapter also discusses various software and hardware optimizations used to accelerate DNN computations on these platforms without impacting application accuracy.

- Chapter 5 describes the design of specialized hardware for DNN processing, with a focus on spatial architectures. It highlights the processing order and resulting data movement in the hardware used to process a DNN and the relationship to a loop nest representation of a DNN. The order of the loops in the loop nest is referred to as the *dataflow*, and it determines how often each piece of data needs to be moved. The limits of the loops in the loop nest describe how to break the DNN workload into smaller pieces, referred to as *tiling/blocking* to account for the limited storage capacity at different levels of the memory hierarchy.

- Chapter 6 presents the process of *mapping* a DNN workload on to a DNN accelerator. It describes the steps required to find an optimized mapping, including enumerating all legal mappings and searching those mappings by employing models that project throughput and energy efficiency.

The third module discusses how additional improvements in efficiency can be achieved either by moving up the stack through the co-design of the algorithms and hardware or down the stack by using mixed signal circuits and new memory or device technology. In the cases where the algorithm is modified, the impact on accuracy must be carefully evaluated.

- Chapter 7 describes how reducing the precision of data and computation can result in increased throughput and energy efficiency. It discusses how to reduce precision using quantization and the associated design considerations, including hardware cost and impact on accuracy.
- Chapter 8 describes how exploiting sparsity in DNNs can be used to reduce the footprint of the data, which provides an opportunity to reduce storage requirements, data movement, and arithmetic operations. It describes various sources of sparsity and techniques to increase sparsity. It then discusses how sparse DNN accelerators can translate sparsity into improvements in energy-efficiency and throughput. It also presents a new abstract data representation that can be used to express and obtain insight about the dataflows for a variety of sparse DNN accelerators.
- Chapter 9 describes how to optimize the structure of the DNN models (i.e., the 'network architecture' of the DNN) to improve both throughput and energy efficiency while trying to minimize impact on accuracy. It discusses both manual design approaches as well as automatic design approaches (i.e., neural architecture search).
- Chapter 10, on advanced technologies, discusses how mixed-signal circuits and new memory technologies can be used to bring the compute closer to the data (e.g., processing in memory) to address the expensive data movement that dominates throughput and energy consumption of DNNs. It also briefly discusses the promise of reducing energy consumption and increasing throughput by performing the computation and communication in the optical domain.

What's New?

This book is an extension of a tutorial paper written by the same authors entitled "Efficient Processing of Deep Neural Networks: A Tutorial and Survey" that appeared in the *Proceedings* *of the IEEE* in 2017 and slides from short courses given at ISCA and MICRO in 2016, 2017, and 2019 (slides available at http://eyeriss.mit.edu/tutorial.html). This book includes recent works since the publication of the tutorial paper along with a more in-depth treatment of topics such as dataflow, mapping, and processing in memory. We also provide updates on the fast-moving field of co-design of DNN models and hardware in the areas of reduced precision, sparsity, and efficient DNN model design. As part of this effort, we present a new way of thinking about sparse representations and give a detailed treatment of how to handle and exploit sparsity. Finally, we touch upon recurrent neural networks, auto encoders, and transformers, which we did not discuss in the tutorial paper.

Scope of book

The main goal of this book is to teach the reader how to tackle the computational challenge of efficiently processing DNNs rather than how to design DNNs for increased accuracy. As a result, this book does not cover training (only touching on it lightly), nor does it cover the theory of deep learning or how to design DNN models (though it discusses how to make them efficient) or use them for different applications. For these aspects, please refer to other references such as Goodfellow's book [2], Amazon's book [3], and Stanford cs231n course notes [4].

Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer
June 2020

Acknowledgments

The authors would like to thank Margaret Martonosi for her persistent encouragement to write this book. We would also like to thank Liane Bernstein, Davis Blalock, Natalie Enright Jerger, Jose Javier Gonzalez Ortiz, Fred Kjolstad, Yi-Lun Liao, Andreas Moshovos, Boris Murmann, James Noraky, Angshuman Parashar, Michael Pellauer, Clément Pit-Claudel, Sophia Shao, Mahmhut Ersin Sinangil, Po-An Tsai, Marian Verhelst, Tom Wenisch, Diana Wofk, Nellie Wu, and students in our "Hardware Architectures for Deep Learning" class at MIT, who have provided invaluable feedback and discussions on the topics described in this book. We would also like to express our deepest appreciation to Robin Emer for her suggestions, support, and tremendous patience during the writing of this book.

As mentioned earlier in the Preface, this book is an extension of an earlier tutorial paper, which was based on tutorials we gave at ISCA and MICRO. We would like to thank David Brooks for encouraging us to do the first tutorial at MICRO in 2016, which sparked the effort that led to this book.

This work was funded in part by DARPA YFA, the DARPA contract HR0011-18-3-0007, the MIT Center for Integrated Circuits and Systems (CICS), the MIT-IBM Watson AI Lab, the MIT Quest for Intelligence, the NSF E2CDA 1639921, and gifts/faculty awards from Nvidia, Facebook, Google, Intel, and Qualcomm.

Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer
June 2020

PART I

Understanding Deep Neural Networks

CHAPTER 1
Introduction

Deep neural networks (DNNs) are currently the foundation for many modern artificial intelligence (AI) applications [5]. Since the breakthrough application of DNNs to speech recognition [6] and image recognition[1] [7], the number of applications that use DNNs has exploded. These DNNs are employed in a myriad of applications from self-driving cars [8], to detecting cancer [9], to playing complex games [10]. In many of these domains, DNNs are now able to exceed human accuracy. The superior accuracy of DNNs comes from their ability to extract high-level features from raw sensory data by using statistical learning on 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. To date, general-purpose compute engines, especially graphics processing units (GPUs), have been the mainstay for much DNN processing. Increasingly, however, in these waning days of Moore's law, there is a recognition that more specialized hardware is needed to keep improving compute performance and energy efficiency [11]. This is especially true in the domain of DNN computations. This book aims to provide an overview of DNNs, the various tools for understanding their behavior, and the techniques being explored to efficiently accelerate their computation.

1.1 BACKGROUND ON DEEP NEURAL NETWORKS

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

1.1.1 ARTIFICIAL INTELLIGENCE AND DEEP NEURAL NETWORKS

DNNs, also referred to as deep learning, are a part of the broad field of AI. AI is the science and engineering of creating intelligent machines that have the ability to 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 AI is illustrated in Figure 1.1.

[1]Image recognition is also commonly referred to as image classification.

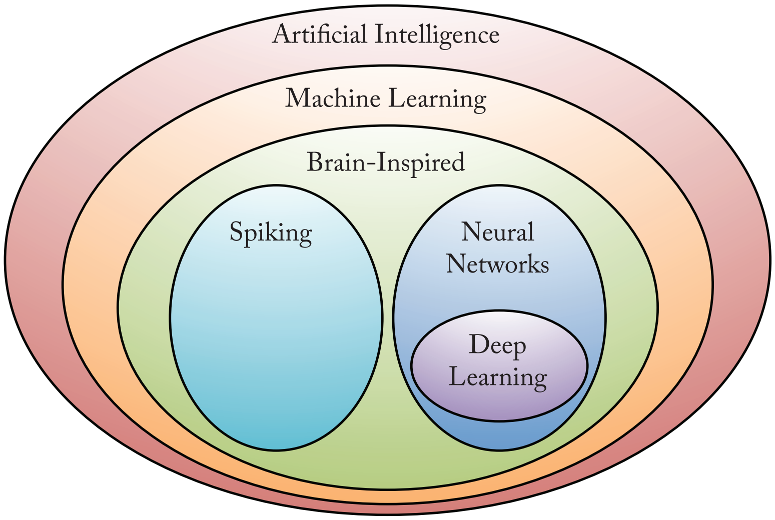


Figure 1.1: Deep learning in the context of artificial intelligence.

Within AI is a large sub-field called machine learning, which was defined in 1959 by Arthur Samuel [12] 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, a single machine learning algorithm simply needs to learn, via a process 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 of for learning and solving problems, it is a natural place to look for inspiration. 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 by a number of elements entering them, called dendrites, and an element leaving them, called an axon, as shown in Figure 1.2. 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 sig-

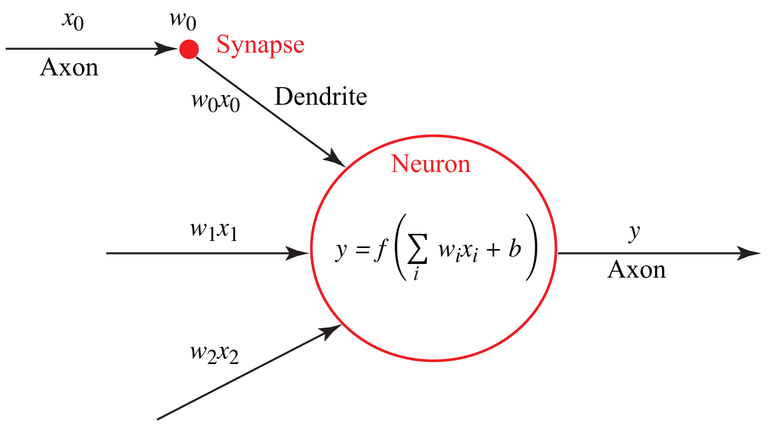


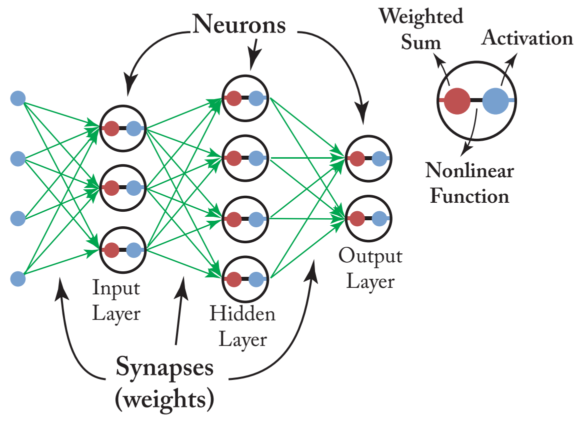
Figure 1.2: Connections to a neuron in the brain. x_i, w_i, $f(\cdot)$, and b are the activations, weights, nonlinear function, and bias, respectively. (Figure adapted from [4].)

nals 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.

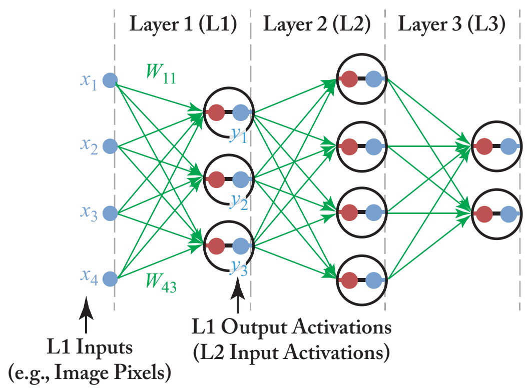
A key characteristic of the synapse is that it can scale the signal (x_i) crossing it, as shown in Figure 1.2. That scaling factor can be referred to as a *weight* (w_i), and the way the brain is believed to learn is through changes to the weights associated with the synapses. Thus, different weights result in different responses to an input. One aspect of learning can be thought of as the adjustment of weights in response to a learning stimulus, while the organization (what might be thought of as the program) of the brain largely 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. The IBM TrueNorth project is an example of work that was inspired by the spiking of the brain [13]. In contrast to spiking computing, another subarea of brain-inspired computing is called *neural networks*, which is the focus of this book.[2]

[2]Note: Recent work using TrueNorth in a stylized fashion allows it to be used to compute reduced precision neural networks [14]. These types of neural networks are discussed in Chapter 7.



(a) Neurons and synapses



(b) Compute weighted sum for each layer

Figure 1.3: Simple neural network example and terminology. (Figure adapted from [4].)

1.1.2 NEURAL NETWORKS AND DEEP NEURAL NETWORKS

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 does not directly output that weighted sum because the expressive power of the cascade of neurons involving only linear operations is just equal to that of a single neuron, which is very limited. Instead, there is a functional operation within the neuron that is performed on the combined inputs. This operation appears to be a nonlinear function that causes a neuron to generate an output only if its combined inputs cross some threshold. Thus, by analogy, neural networks apply a nonlinear function to the weighted sum of the input values.[3] These nonlinear functions are inspired by biological functions, but are not meant to emulate the brain. We look at some of those nonlinear functions in Section 2.3.3.

Figure 1.3a shows a diagram of a three-layer (non-biological) neural network. The neurons in the input layer receive some values, compute their weighted sums followed by the nonlinear function, and propagate the outputs to the neurons in the middle layer of the network, which is also frequently called a "hidden layer." A neural network can have more than one hidden layer, and the outputs from the hidden layers ultimately propagate to the output layer, which computes 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 Figure 1.3a. We will use the activation/weight nomenclature in this book.

[3]Without a nonlinear function, multiple layers could be collapsed into one.

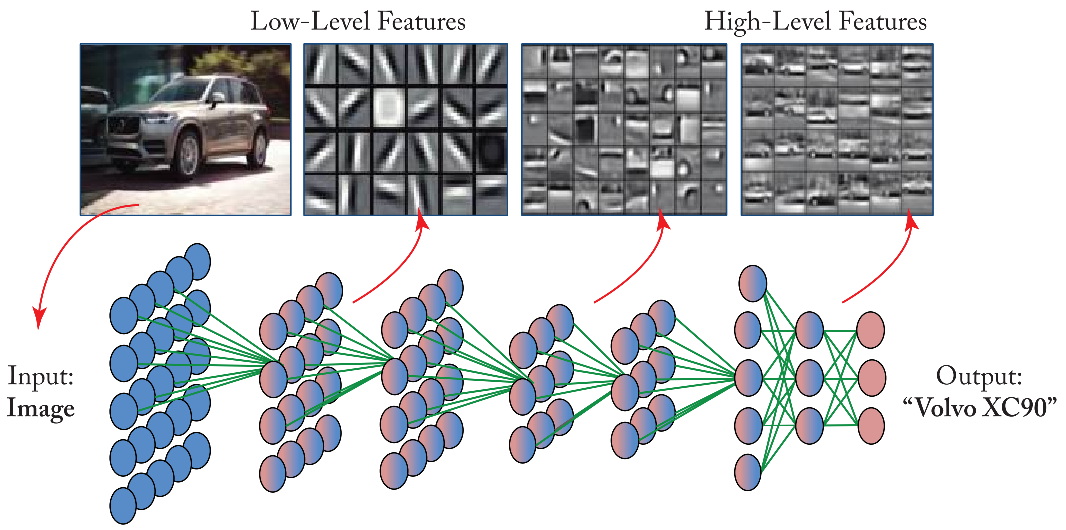


Figure 1.4: Example of image classification using deep neural networks. (Figure adapted from [15].) Note that the features go from low level to high level as we go deeper into the network.

Figure 1.3b shows an example of the computation at layer 1: $y_j = f(\sum_{i=1}^{4} W_{ij} \times x_i + b_j)$, where W_{ij}, x_i, and y_j are the weights, input activations, and output activations, respectively, and $f(\cdot)$ is a nonlinear function described in Section 2.3.3. The bias term b_j is omitted from Figure 1.3b for simplicity. In this book, we will use the color green to denote weights, blue to denote activations, and red to denote weighted sums (or partial sums, which are further accumulated to become the final weighted sums).

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 5 to more than a 1,000. In this book, 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, as shown in Figure 1.4. 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. In 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. Finally, given all this information, the network provides a probability that these high-level fea-

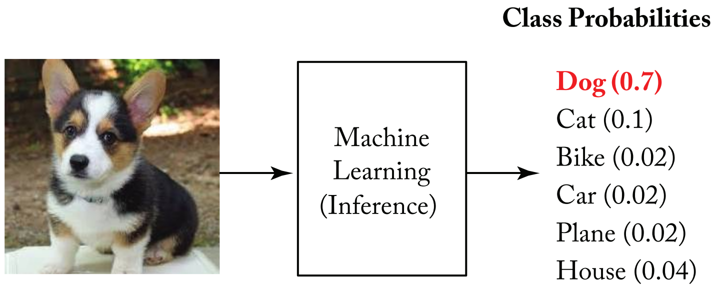


Figure 1.5: Example of an image classification task. The machine learning platform takes in an image and outputs the class probabilities for a predefined set of classes.

tures comprise a particular object or scene. This deep feature hierarchy enables DNNs to achieve superior performance in many tasks.

1.2 TRAINING VERSUS INFERENCE

Since DNNs are an instance of machine learning algorithms, 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 biases) 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 Figure 1.5, as a driving example for training and using a DNN. When we perform inference using a DNN, the input is image and the output is a vector of values representing the *class probabilities*. There is one value for each object class, and the class with the highest value indicates the most likely (predicted) class of object in the image. The overarching goal for training a DNN is to determine the weights that maximize the probability of the correct class and minimize the probabilities of the incorrect classes. The correct class is generally known, as it is often defined in the training set. The gap between the ideal correct probabilities and the probabilities 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 (hill-descending) optimization process called gradient descent. In gradient descent, a weight is updated by a scaled version of the partial derivative of the loss with respect to the weight (i.e.,

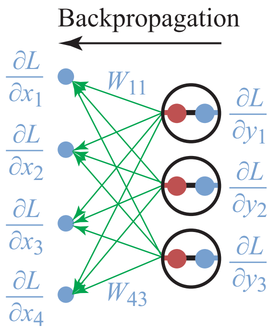
Sidebar: Key steps in training

Here, we will provide a very brief summary of the key steps of training and deploying a model. For more details, we recommend the reader refer to more comprehensive references such as [2]. First, we collect a labeled dataset and divide the data into subsets for training and testing. Second, we use the training set to train a model so that it can learn the weights for a given task. After achieving adequate accuracy on the training set, the ultimate quality of the model is determined by how accurately it performs on unseen data. Therefore, in the third step, we test the trained model by asking it to predict the labels for a test set that it has never seen before and compare the prediction to the ground truth labels. *Generalization* refers to how well the model maintains the accuracy between training and unseen data. If the model does not generalize well, it is often referred to as *overfitting*; this implies that the model is fitting to the noise rather than the underlying data structure that we would like it to learn. One way to combat overfitting is to have a large, diverse dataset; it has been shown that accuracy increases logarithmically as a function of the number of training examples [16]. Section 2.6.3 will discuss various popular datasets used for training. There are also other mechanisms that help with generalization including *Regularization*. It adds constraints to the model during training such as smoothness, number of parameters, size of the parameters, prior distribution or structure, or randomness in the training using dropout [17]. Further partitioning the training set into training and *validation* sets is another useful tool. Designing a DNN requires determining (tuning) a large number of hyperparameters such as the size and shape of a layer or the number of layers. Tuning the hyperparameters based on the test set may cause overfitting to the test set, which results in a misleading evaluation of the true performance on unseen data. In this circumstance, the validation set can be used instead of the test set to mitigate this problem. Finally, if the model performs sufficiently well on the test set, it can be deployed on unlabeled images.

updated to $w_{ij}^{t+1} = w_{ij}^{t} - \alpha \frac{\partial L}{\partial w_{ij}}$, where α is called the learning rate[4]). 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.

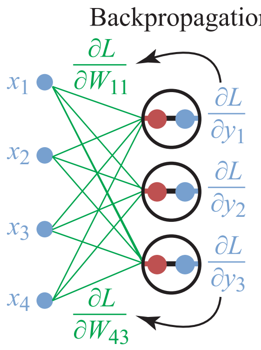
An efficient way to compute the partial derivatives of the gradient is through a process called *backpropagation*. Backpropagation, which is a computation derived from the *chain rule* of

[4]A large learning rate increases the step size applied at each iteration, which can help speed up the training, but may also result in overshooting the minimum or cause the optimization to not converge. A small learning rate decreases the step size applied at each iteration which slows down the training, but increases likelihood of convergence. There are various methods to set the learning rate such as ADAM [18], etc. Finding the best the learning rate is one of the key challenges in training DNNs.



Backpropagation
$\frac{\partial L}{\partial x_1}$
$\frac{\partial L}{\partial x_2}$
$\frac{\partial L}{\partial x_3}$
$\frac{\partial L}{\partial x_4}$
W_{11}
W_{43}
$\frac{\partial L}{\partial y_1}$
$\frac{\partial L}{\partial y_2}$
$\frac{\partial L}{\partial y_3}$

(a) Compute the gradient of the loss relative to the layer inputs ($\frac{\partial L}{\partial x_i} = \sum_j w_{ij} \frac{\partial L}{\partial y_j}$)



Backpropagation
$\frac{\partial L}{\partial W_{11}}$
$\frac{\partial L}{\partial W_{43}}$
x_1
x_2
x_3
x_4
$\frac{\partial L}{\partial y_1}$
$\frac{\partial L}{\partial y_2}$
$\frac{\partial L}{\partial y_3}$

(b) Compute the gradient of the loss relative to the weights ($\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial y_j} x_i$)

Figure 1.6: An example of backpropagation through a neural network.

calculus, operates by passing values backward 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 Figure 1.6 [19].[5] Thus, techniques for efficiently performing inference can sometimes be useful for performing training. There are, however, some important additional considerations to note. First, backpropagation requires intermediate outputs of the network to be preserved for the backward computation, thus training has increased storage requirements. Second, due to the gradients use for hill-climbing (hill-descending), the precision requirement for training is generally higher than inference. Thus, many of the reduced precision techniques discussed in Chapter 7 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 inputs is computed before a single pass of weight updates is performed. This is called *batching*, which helps to speed up and stabilize the process.[6]

[5]To backpropagate through each layer: (1) compute the gradient of the loss relative to the weights, $\frac{\partial L}{\partial w_{ij}}$, from the layer inputs (i.e., the forward activations, x_i) and the gradients of the loss relative to the layer outputs, $\frac{\partial L}{\partial y_j}$; and (2) compute the gradient of the loss relative to the layer inputs, $\frac{\partial L}{\partial x_i}$, from the layer weights, w_{ij}, and the gradients of the loss relative to the layer outputs, $\frac{\partial L}{\partial y_j}$.

[6]There are various forms of gradient decent which differ in terms of how frequently to update the weights. *Batch Gradient Descent* updates the weights after computing the loss on the entire training set, which is computationally expensive and requires significant storage. *Stochastic Gradient Descent* update weights after computing loss on a single training example and the examples are shuffled after going through the entire training set. While it is fast, looking at a single example can be noisy and cause the weights to go in the wrong direction. Finally, *Mini-batch Gradient Descent* divides the training set into smaller sets called mini-batches, and updates weights based on the loss of each mini-batch (commonly referred to simply as "batch"); this approach is most commonly used. In general, each pass through the entire training set is referred to as an *epoch*.

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 no training samples are labeled. Essentially, the goal is to find the structure or clusters in the data. *Semi-supervised learning* falls 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 the 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 (often referred to as an episode).

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 book 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.

1.3 DEVELOPMENT HISTORY

Although neural networks 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 handwritten digit recognition [20].[7] Such systems are widely used by ATMs for digit recognition on checks. The early 2010s have seen a blossoming of DNN-based applications, with highlights such as Microsoft's speech recognition system in 2011 [6] and the AlexNet DNN for image recognition in 2012 [7]. A brief chronology of deep learning is shown in Figure 1.7.

The deep learning successes of the early 2010s are believed to be due to 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 up to a billion images per day, Walmart creates 2.5 Petabytes of customer data hourly and YouTube has over 300 hours of video uploaded every minute. As a result, these and many other businesses have a huge amount of data to train their algorithms.

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.

[7]In the early 1960s, single neuron systems built out of analog logic were used for adaptive filtering [21, 22].

DNN Timeline

- 1940s: Neural networks were proposed
- 1960s: Deep neural networks were proposed
- 1989: Neural networks for recognizing hand-written 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…)

Figure 1.7: A concise history of neural networks. "Deep" refers to the number of layers in the network.

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 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 [23]. 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 Figure 1.5. The training set consists of 1.2 million images, each of which is labeled with one of a thousand 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.

Figure 1.8 shows the performance of the best entrants in the ImageNet contest over a number of years. 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 DNN approach, named AlexNet, and reduced the error rate by approximately 10 percentage points [7]. Their accomplishment inspired an outpouring of deep learning algorithms that have resulted in a steady stream of improvements.

In conjunction with the trend toward using 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 use of GPUs reflects the almost complete switch from traditional computer vision approaches to deep learning-based approaches for the competition.

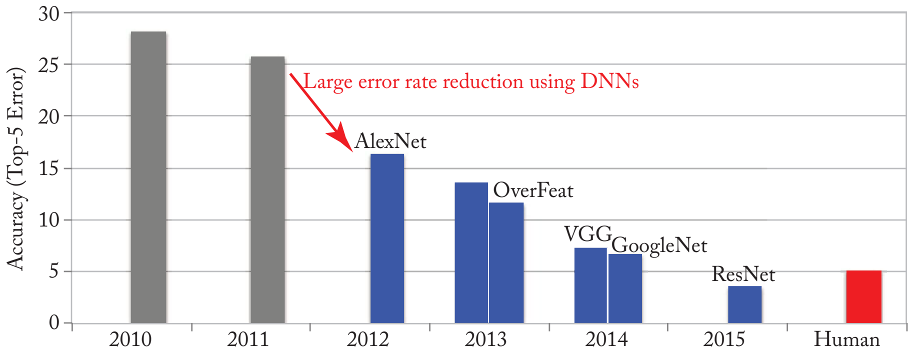


Figure 1.8: Results from the ImageNet Challenge [23].

In 2015, the ImageNet winning entry, ResNet [24], exceeded human-level accuracy with a Top-5 error rate[8] 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.

1.4 APPLICATIONS OF DNNs

Many domains can benefit from DNNs, ranging from entertainment to medicine. In this section, we will provide examples of areas where DNNs are currently making an impact and highlight emerging areas where DNNs may make an impact in the future.

- **Image and Video:** Video is arguably the biggest of big data. It accounts for over 70% of today's Internet traffic [25]. For instance, over 800 million hours of video is collected daily worldwide for video surveillance [26]. 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 [23], object localization and detection [27], image segmentation [28], and action recognition [29].

- **Speech and Language:** DNNs have significantly improved the accuracy of speech recognition [30] as well as many related tasks such as machine translation [6], natural language processing [31], and audio generation [32].

- **Medicine and Health Care:** DNNs have played an important role in genomics to gain insight into the genetics of diseases such as autism, cancers, and spinal muscular atrophy [33–

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

36]. They have also been used in medical imaging such as detecting skin cancer [9], brain cancer [37], and breast cancer [38].

- **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 [39]. DNNs have surpassed human level accuracy in playing games such as Atari [40], Go [10], and StarCraft [41], where an exhaustive search of all possibilities is not feasible due to the immense number of possible moves.

- **Robotics:** DNNs have been successful in the domain of robotic tasks such as grasping with a robotic arm [42], motion planning for ground robots [43], visual navigation [8, 44], control to stabilize a quadcopter [45], and driving strategies for autonomous vehicles [46].

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 [47]. 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.

1.5 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[9] and significant computational resources for multiple weight-update iterations. In many cases, training a DNN model still takes several hours to multiple days (or weeks or months!) 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., Internet of Things (IoT) or mobile). In many applications, it is desirable to have the DNN inference processing at the edge 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 these applications.[10] Speech recognition allows us to seamlessly interact with electronic devices, such

[9]One of the major drawbacks of DNNs is their need for large datasets to prevent overfitting during training.

[10]As a reference, running a DNN on an embedded devices is estimated to consume several orders of magnitude higher energy per pixel than video compression, which is a common form of processing near image sensor [48].

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. Some work have even considered partitioning the processing between the cloud and edge at a per layer basis in order to improve performance [49]. However, considerations related to dependency on connectivity, privacy, and security augur for keeping computation at the edge. Many of the embedded platforms that perform DNN inference have stringent requirements on energy consumption, compute and memory cost limitations; efficient processing of DNNs has become of prime importance under these constraints.

CHAPTER 2

Overview of Deep Neural Networks

Deep Neural Networks (DNNs) come in a wide variety of shapes and sizes depending on the application.[1] 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.

In this chapter, we will describe the key building blocks for DNNs. As there are *many* different types of DNNs [50], we will focus our attention on those that are most widely used. We will begin by describing the salient characteristics of commonly used DNN layers in Sections 2.1 and 2.2. We will then describe popular DNN layers and how these layers can be combined to form various types of DNNs in Section 2.3. Section 2.4 will provide a detailed discussion on convolutional neural networks (CNNs), since they are widely used and tend to provide many opportunities for efficient DNN processing. It will also highlight various popular CNN models that are often used as workloads for evaluating DNN hardware accelerators. Next, in Section 2.5, we will briefly discuss other types of DNNs and describe how they are similar to and differ from CNNs from a workload processing perspective (e.g., data dependencies, types of compute operations, etc.). Finally, in Section 2.6, we will discuss the various DNN development resources (e.g., frameworks and datasets), which researchers and practitioners have made available to help enable the rapid progress in DNN model and hardware research and development.

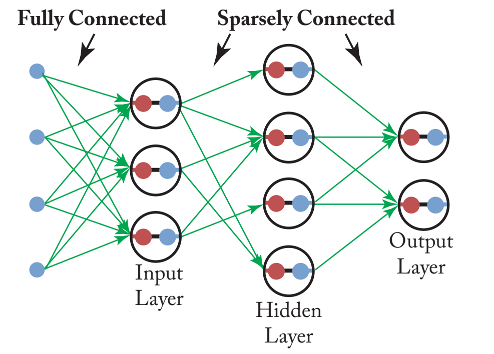
2.1 ATTRIBUTES OF CONNECTIONS WITHIN A LAYER

As discussed in Chapter 1, DNNs are composed of several processing layers, where in most layers the main computation is a weighted sum. There are several different types of layers, which primarily differ in terms of how the inputs and outputs are connected *within* the layers.

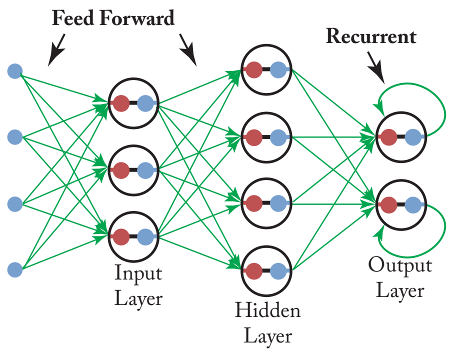
There are two main attributes of the connections within a layer:

1. The connection pattern between the input and output activations, as shown in Figure 2.1a: if a layer has the attribute that every input activation is connected to every output, then we

[1]The DNN research community often refers to the shape and size of a DNN as its “network architecture.” However, to avoid confusion with the use of the word “architecture” by the hardware community, we will talk about “DNN models” and their shape and size in this book.



(a) Fully connected versus sparsely connected



(b) Feed-forward versus feed-back (recurrent) connections

Figure 2.1: Properties of connections in DNNs (Figure adapted from [4]).

call that layer *fully connected*. On the other hand, if a layer has the attribute that only a subset of inputs are connected to the output, then we call that layer *sparsely connected*. Note that the weights associated with these connections can be zero or non-zero; if a weight happens to be zero (e.g., as a result of training), it does not mean there is no connection (i.e., the connection still exists).

For sparsely connected layers, a sub attribute is related to the structure of the connections. Input activations may connect to any output activation (i.e., global), or they may only connect to output activations in their neighborhood (i.e., local). The consequence of such local connections is that each output activation is a function of a restricted window of input activations, which is referred to as the *receptive field*.

2. The value of the weight associated with each connection: the most general case is that the weight can take on any value (e.g., each weight can have a unique value). A more restricted case is that the same value is shared by multiple weights, which is referred to as *weight sharing*.

Combinations of these attributes result in many of the common layer types. Any layer with the fully connected attribute is called a fully connected layer (FC layer). In order to distinguish the attribute from the type of layer, in this chapter, we will use the term FC layer as distinguished from the fully connected attribute. However, in subsequent chapters we will follow the common practice of using the terms interchangeably. Another widely used layer type is the convolutional (CONV) layer, which is locally, sparsely connected with weight sharing.[2] The computation in FC and CONV layers is a weighted sum. However, there are other computations that might be

[2]CONV layers use a specific type of weight sharing, which will be described in Section 2.4.

performed and these result in other types of layers. We will discuss FC, CONV, and these other layers in more detail in Section 2.3.

2.2 ATTRIBUTES OF CONNECTIONS BETWEEN LAYERS

Another attribute is the connections from the output of one layer to the input of *another* layer, as shown in Figure 2.1b. The output can be connected to the input of the next layer in which case the connection is referred to as *feed forward*. With feed-forward connections, all of the computation is performed as a sequence of operations on the outputs of a previous layer.[3] It has no memory and the output for an input is always the same irrespective of the sequence of inputs previously given to the network. DNNs that contain feed-forward connections are referred to as *feed-forward* networks. Examples of these types of networks include multi-layer perceptrons (MLPs), which are DNNs that are composed entirely of feed-forward FC layers and convolutional neural networks (CNNs), which are DNNs that contain both FC and CONV layers. CNNs, which are commonly used for image processing and computer vision, will be discussed in more detail in Section 2.4.

Alternatively, the output can be fed back to the input of its own layer in which case the connection is often referred to as *recurrent*. With recurrent connections, the output of a layer is a function of both the current and prior input(s) to the layer. This creates a form of memory in the DNN, which allows long-term dependencies to affect the output. DNNs that contain these connections are referred to as *recurrent* neural networks (RNNs), which are commonly used to process sequential data (e.g., speech, text), and will be discussed in more detail in Section 2.5.

2.3 POPULAR TYPES OF LAYERS IN DNNs

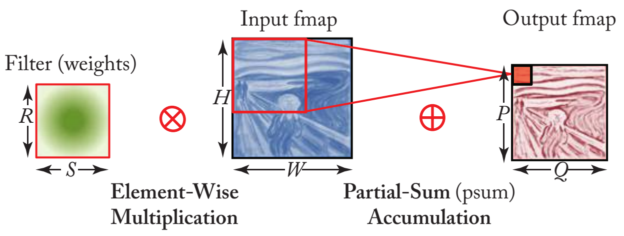
In this section, we will discuss the various popular layers used to form DNNs. We will begin by describing the CONV and FC layers whose main computation is a weighted sum, since that tends to dominate the computation cost in terms of both energy consumption and throughput. We will then discuss various layers that can optionally be included in a DNN and do not use weighted sums such as nonlinearity, pooling, and normalization.

These layers can be viewed as primitive layers, which can be combined to form compound layers. Compound layers are often given names as a convenience, when the same combination of primitive layer are frequently used together. In practice, people often refer to either primitive or compound layers as just layers.

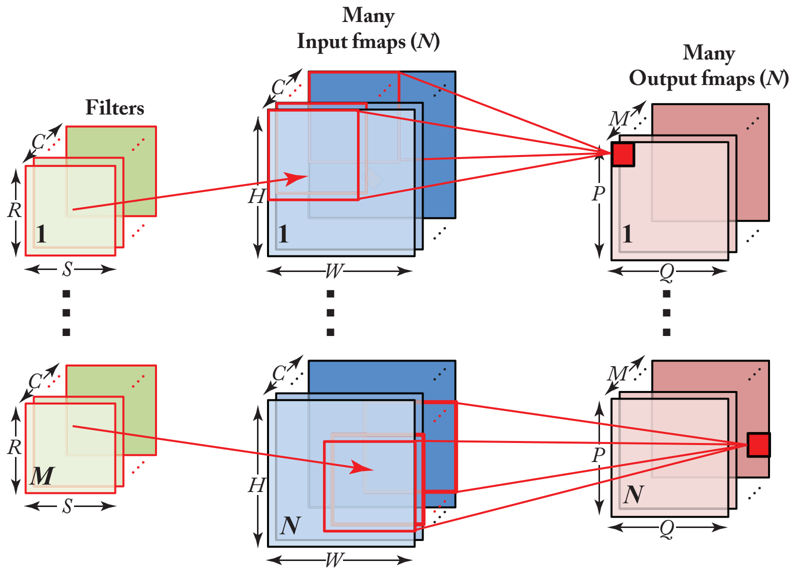
2.3.1 CONV LAYER (CONVOLUTIONAL)

CONV layers are primarily composed of high-dimensional convolutions, as shown in Figure 2.2. In this computation, the input activations of a layer are structured as a 3-D *input feature map*

[3]Connections can come from the immediately preceding layer or an earlier layer. Furthermore, connections from a layer can go to multiple later layers.



(a) 2-D convolution in traditional image processing



(b) High-dimensional convolutions in CNNs

Figure 2.2: Dimensionality of convolutions. (a) Shows the traditional 2-D convolution used in image processing. (b) Shows the high dimensional convolution used in CNNs, which applies a 2-D convolution on each channel.

(ifmap), where the dimensions are the height (H), width (W), and number of input channels (C). The weights of a layer are structured as a 3-D filter, where the dimensions are the height (R), width (S), and number of input channels (C). Notice that the number of channels for the input feature map and the filter are the same. For each input channel, the input feature map undergoes a 2-D convolution (see Figure 2.2a) with the corresponding channel in the filter. The results of the convolution at each point are summed across all the input channels to generate the output partial sums. In addition, a 1-D (scalar) bias can be added to the filtering results, but some recent networks [24] remove its usage from parts of the layers. The results of this

Table 2.1: Shape parameters of a CONV/FC layer

Shape Parameter	Description
N	Batch size of 3-D fmaps
M	Number of 3-D filters / number of channels of ofmap (output channels)
C	Number of channels of filter / ifmap (input channels)
H/W	Ifmap spatial height/width
R/S	Filter spatial height/width (= H/W in FC)
P/Q	Ofmap spatial height/width (= 1 in FC)

computation are the output partial sums that comprise one channel of the *output feature map* (ofmap).[4] Additional 3-D filters can be used on the same input feature map to create additional output channels (i.e., applying M filters to the input feature map generates M output channels in the output feature map). Finally, multiple input feature maps (N) may be processed together as a *batch* to potentially improve reuse of the filter weights.

Given the shape parameters in Table 2.1,[5] the computation of a CONV layer is defined as:

$$\begin{aligned}
&\mathbf{o}[n][m][p][q] = \Big(\sum_{c=0}^{C-1}\sum_{r=0}^{R-1}\sum_{s=0}^{S-1}\mathbf{i}[n][c][Up+r][Uq+s]\times\mathbf{f}[m][c][r][s]\Big)+\mathbf{b}[m],\\
&0\le n<N, 0\le m<M, 0\le p<P, 0\le q<Q,\\
&P=(H-R+U)/U, Q=(W-S+U)/U.
\end{aligned}\tag{2.1}$$

o, **i**, **f**, and **b** are the tensors of the ofmaps, ifmaps, filters, and biases, respectively. U is a given stride size.

Figure 2.2b shows a visualization of this computation (ignoring biases). As much as possible, we will adhere to the following coloring scheme in this book.

- **Blue**: input activations belonging to an input feature map.
- **Green**: weights belonging to a filter.

[4]For simplicity, in this chapter, we will refer to an array of partial sums as an output feature map. However, technically, the output feature map would be composed the values of the partial sums *after* they have gone through a nonlinear function (i.e., the output activations).

[5]In some literature, K is used rather than M to denote the number of 3-D filters (also referred to a kernels), which determines the number of output feature map channels. We opted not to use K to avoid confusion with yet other communities that use it to refer to the number of dimensions. We also have adopted the convention of using P and Q as the dimensions of the output to align with other publications and since our prior use of E and F caused an alias with the use of "F" to represent filter weights. Note that some literature also use X and Y to denote the spatial dimensions of the input rather than W and H.

- **Red**: partial sums—Note: since there is no formal term for an array of partial sums, we will sometimes label an array of partial sums as an output feature map and color it red (even though, technically, output feature maps are composed of activations derived from partial sums that have passed through a nonlinear function and therefore should be blue).

Returning to the CONV layer calculation in Equation (2.1), one notes that the operands (i.e., the ofmaps, ifmaps, and filters) have many dimensions. Therefore, these operands can be viewed as *tensors* (i.e., high-dimension arrays) and the computation can be treated as a tensor algebra computation where the computation involves performing binary operations (e.g., multiplications and additions forming dot products) between tensors to produce new tensors. Since the CONV layer can be viewed as a tensor algebra operation, it is worth noting that an alternative representation for a CONV layer can be created using the *tensor index notation* found in [51], which describes a compiler for sparse tensor algebra computations.[6] The tensor index notation provides a compact way to describe a kernel's functionality. For example, in this notation matrix multiply $Z = AB$ can be written as:

$$\mathbf{Z}_{ij} = \sum_k \mathbf{A}_{ik}\mathbf{B}_{kj}. \tag{2.2}$$

That is, the output point (i, j) is formed by taking a dot product of k values along the i-th row of A and the j-th column of B.[7] Extending this notation to express computation on the index variables (by putting those calculations in parenthesis) allows a CONV layer in tensor index notation to be represented quite concisely as:

$$\mathbf{O}_{nmpq} = (\sum_{crs} \mathbf{I}_{nc(Up+r)(Uq+s)}\mathbf{F}_{mcrs}) + \mathbf{b}_m. \tag{2.3}$$

In this calculation, each output at a point (n, m, p, q) is calculated as a dot product taken across the index variables c, r, and s of the specified elements of the input activation and filter weight tensors. Note that this notation attaches no significance to the order of the index variables in the summation. The relevance of this will become apparent in the discussion of dataflows (Chapter 5) and mapping computations onto a DNN accelerator (Chapter 6).

Finally, to align the terminology of CNNs with the generic DNN,

- filters are composed of weights (i.e., synapses), and
- input and output feature maps (ifmaps, ofmaps) are composed of input and output activations (partial sums after application of a nonlinear function) (i.e., input and output neurons).

[6]Note that many of the values in the CONV layer tensors are zero, making the tensors *sparse*. The origins of this sparsity, and approaches for performing the resulting sparse tensor algebra, are presented in Chapter 8.

[7]Note that Albert Einstein popularized a similar notation for tensor algebra which omits any explicit specification of the summation variable.

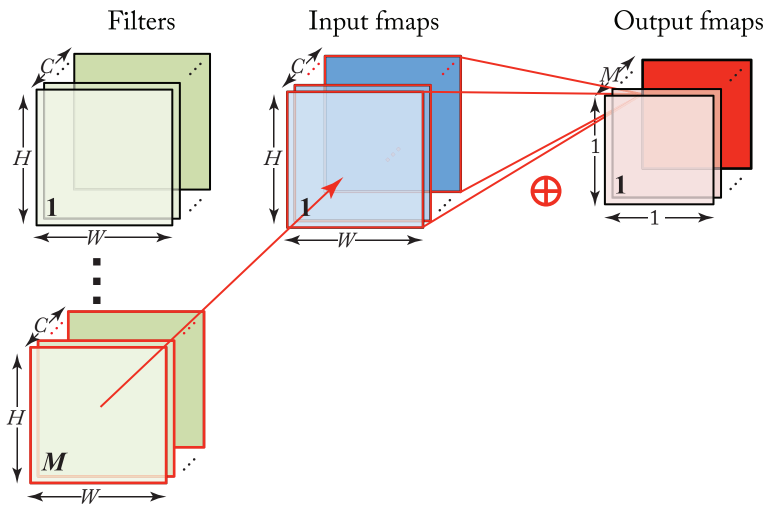


Figure 2.3: Fully connected layer from convolution point of view with $H = R$, $W = S$, $P = Q = 1$, and $U = 1$.

2.3.2 FC LAYER (FULLY CONNECTED)

In an FC layer, every value in the output feature map is a weighted sum of every input value in the input feature map (i.e., it is fully connected). Furthermore, FC layers typically do not exhibit weight sharing and as a result the computation tends to be memory-bound. FC layers are often processed in the form of a matrix multiplication, which will be explained in Chapter 4. This is the reason while matrix multiplication is often associated with DNN processing.

An FC layer can also be viewed as a special case of a CONV layer. Specifically, a CONV layer where the filters are of the same size as the input feature maps. Therefore, it does not have the local, sparsely connected with weight sharing property of CONV layers. Therefore, Equation (2.1) still holds for the computation of FC layers with a few additional constraints on the shape parameters: $H = R$, $W = S$, $P = Q = 1$, and $U = 1$. Figure 2.3 shows a visualization of this computation and in the tensor index notation from Section 2.3.1 it is:

$$\mathbf{O}_{nm} = \sum_{chw} \mathbf{I}_{nchw} \mathbf{F}_{mchw}. \tag{2.4}$$

2.3.3 NONLINEARITY

A nonlinear activation function is typically applied after each CONV or FC layer. Various nonlinear functions are used to introduce nonlinearity into the DNN, as shown in Figure 2.4. These include historically conventional nonlinear functions such as sigmoid or hyperbolic tangent. These were popular because they facilitate mathematical analysis/proofs. The rectified linear unit

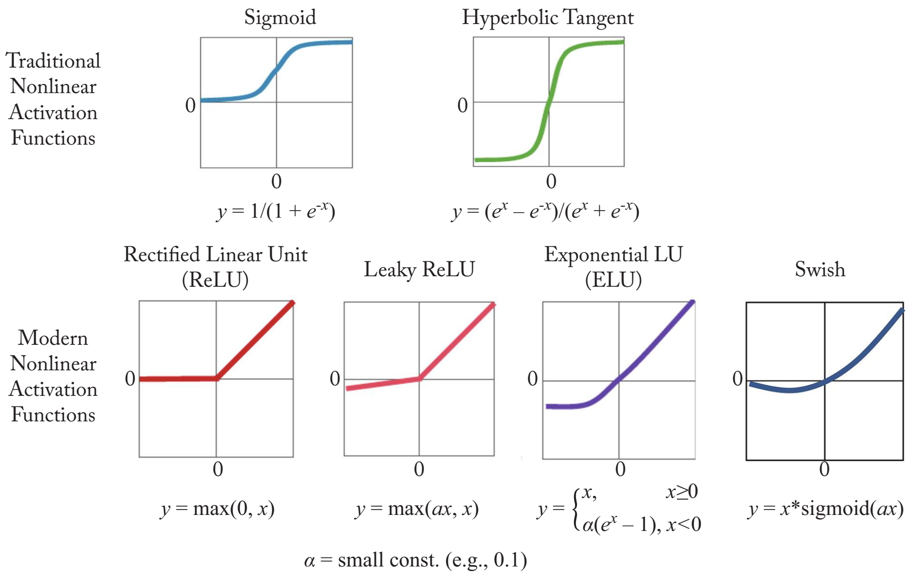


Figure 2.4: Various forms of nonlinear activation functions. (Figure adapted from [62].)

(ReLU) [52] has become popular in recent years due to its simplicity and its ability to enable fast training, while achieving comparable accuracy.[8] Variations of ReLU, such as leaky ReLU [53], parametric ReLU [54], exponential LU [55], and Swish [56] have also been explored for improved accuracy. Finally, a nonlinearity called maxout, which takes the maximum value of two intersecting linear functions, has shown to be effective in speech recognition tasks [57, 58].

2.3.4 POOLING AND UNPOOLING

There are a variety of computations that can be used to change the spatial resolution (i.e., H and W or P and Q) of the feature map depending on the application. For applications such as image classification, the goal is to summarize the entire image into one label; therefore, reducing the spatial resolution may be desirable. Networks that reduce input into a sparse output are often referred to as *encoder networks*. For applications such as semantic segmentation, the goal is to assign a label to each pixel in the image;[9] as a result, increasing the spatial resolution may be desirable. Networks that expand input into a dense output are often referred to as *decoder networks*.

Reducing the spatial resolution of a feature map is referred to as *pooling* or more generically downsampling. Pooling, which is applied to each channel separately, enables the network to be

[8]In addition to being simple to implement, ReLU also increases the sparsity of the output activations, which can be exploited by a DNN accelerator to increase throughput, reduce energy consumption and reduce storage cost, as described in Section 8.1.1.

[9]In the literature, this is often referred to *dense* prediction.

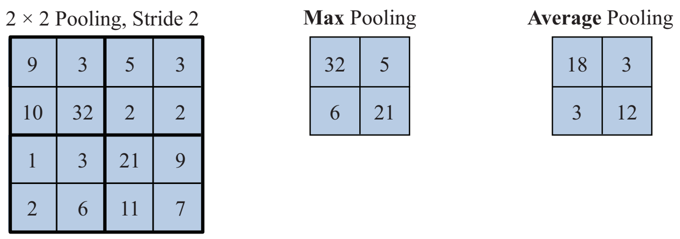
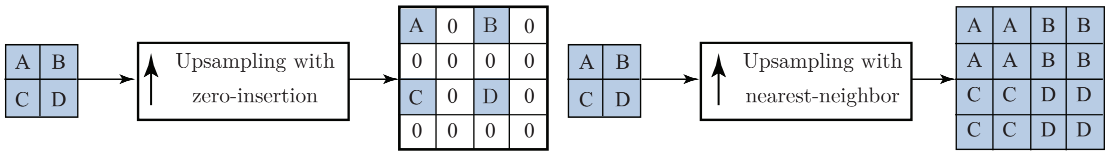


Figure 2.5: Various forms of pooling.



(a) Zero insertion (unpooling) (b) Nearest neighbor

Figure 2.6: Various forms of unpooling/upsampling. (Figures adapted from [64].)

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. Pooling can be parameterized based on the size of its receptive field (e.g., 2×2) and pooling operation (e.g., max or average), as shown in Figure 2.5. 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 spatial resolution of the representation (i.e., feature map). Pooling is usually performed *after* the nonlinearity.

Increasing the spatial resolution of a feature map is referred to as *unpooling* or more generically as upsampling. Commonly used forms of upsampling include inserting zeros between the activations, as shown in Figure 2.6a (this type of upsampling is commonly referred to as unpooling[10]), interpolation using nearest neighbors [63, 64], as shown in Figure 2.6b, and interpolation with bilinear or bicubic filtering [65]. Upsampling is usually performed *before* the CONV or FC layer. Upsampling can introduce structured sparsity in the input feature map that can be exploited for improved energy efficiency and throughput, as described in Section 8.1.1.

2.3.5 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 normal-

[10]There are two versions of unpooling: (1) zero insertion is applied in a regular pattern, as shown in Figure 2.6a [60]—this is most commonly used; and (2) unpooling is paired with a max pooling layer, where the location of the max value during pooling is stored, and during unpooling the location of the non-zero value is placed in the location of the max value before pooling [61].

ized 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 Equation (2.5), where the parameters (γ, β) are learned from training [66]:[11],[12]

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta, \tag{2.5}$$

where ϵ is a small constant to avoid numerical problems.

Prior to the wide adoption of BN, local response normalization (LRN) [7] 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 is usually performed after the nonlinear function, BN is usually performed between the CONV or FC layer and the nonlinear function. If BN is performed immediately after the CONV or FC layer, its computation can be folded into the weights of the CONV or FC layer resulting in no additional computation for inference.

2.3.6 COMPOUND LAYERS

The above primitive layers can be combined to form compound layers. For instance, *attention* layers are composed of matrix multiplications and feed-forward, fully connected layers [68]. Attention layers have become popular for processing a wide range of data including language and images and are commonly used in a type of DNNs called Transformers. We will discuss transformers in more detail in Section 2.5. Another example of a compound layer is the *up-convolution* layer [60], which performs zero-insertion (unpooling) on the input and then applies a convolutional layer.[13] Up-convolution layers are typically used in DNNs such as General Adversarial Networks (GANs) and Auto Encoders (AEs) that process image data. We will discuss GANs and AEs in more detail in Section 2.5.

2.4 CONVOLUTIONAL NEURAL NETWORKS (CNNs)

CNNs are a common form of DNNs that are composed of multiple CONV layers, as shown in Figure 2.7. In such networks, each layer generates a successively higher-level abstraction of

[11]It has been recently reported that the reason batch normalization enables faster and more stable training is due to the fact that it makes the optimization landscape smoother resulting in more predictive and stable behavior of the gradient [67]; this is in contrast to the popular belief that batch normalization stabilizes the distribution of the input across layers. Nonetheless, batch normalization continues to be widely used for training and thus needs to be supported during inference.

[12]During training, parameters σ and μ are computed per batch, and γ and β are updated per batch based on the gradient; therefore, training for different batch sizes will result in different σ and μ parameters, which can impact accuracy. Note that each channel has its own set of σ, μ, γ, and β parameters. During inference, all parameters are fixed, where σ and μ are computed from the entire training set. To avoid performing an extra pass over the entire training set to compute σ and μ, σ and μ are usually implemented as the running average of the per batch σ and μ computed during training.

[13]Note variants of the up CONV layer with different types of upsampling include deconvolution layer, sub-pixel or fractional convolutional layer, transposed convolutional layer, and backward convolution layer [69].

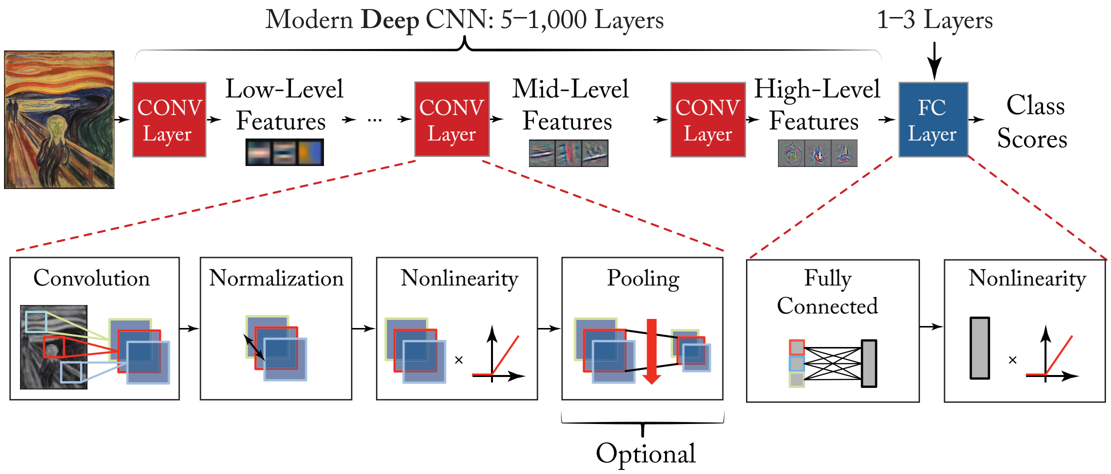


Figure 2.7: Convolutional Neural Networks.

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. CNNs are widely used in a variety of applications including image understanding [7], speech recognition [70], game play [10], robotics [42], etc. This book will focus on its use in image processing, specifically for the task of image classification [7]. Modern CNN models for image classification typically have 5 [7] to more than a 1,000 [24] CONV layers. A small number, e.g., 1 to 3, of FC layers are typically applied after the CONV layers for classification purposes.

2.4.1 POPULAR CNN MODELS

Many CNN models have been developed over the past two decades. Each of these models are different 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 accelerator, as discussed in Chapter 3.

In this section, we will give an overview of various popular CNNs such as LeNet [71] as well as those that competed in and/or won the ImageNet Challenge [23], as shown in Figure 1.8, most of whose models with pre-trained weights are publicly available for download; the CNN models are summarized in Table 2.2. Two results for Top-5 error 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 CNN 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

Table 2.2: Summary of popular CNNs [7, 24, 71, 73, 74]. †Accuracy is measured based on Top-5 error on ImageNet [23] using multiple crops. ‡This version of LeNet-5 has 431k weights for the filters and requires 2.3M MACs per image, and uses ReLU rather than sigmoid.

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
Number of CONV layers	2	5	5	13	57	53
Depth in number of CONV layers	2	5	5	13	21	49
Filter sizes	5	3, 5, 11	2, 5, 11	3	1, 3, 5, 7	1, 3, 7
Number of channels	1, 20	3–256	3–1,024	3–512	3–832	3–2,048
Number of filters	20, 50	96–384	96–1,024	64–512	16–384	64–2,048
Stride	1	1, 4	1, 4	1	1, 2	1, 2
Weights	2.6 k	2.3 M	16 M	14.7 M	6.0 M	23.5 M
MACs	283 k	666 M	2.67 G	15.3 G	1.43 G	3.86 G
Number of FC layers	2	3	3	3	1	1
Filter sizes	1, 4	1, 6	1, 6, 12	1, 7	1	1
Number of channels	50, 500	256–4,096	1,024–4,096	512–4,096	1,024	2,048
Number of filters	10, 500	1,000–4,096	1,000–4,096	1,000–4,096	1,000	1,000
Weights	58 k	58.6 M	130 M	124 M	1 M	2 M
MACS	58 K	58.6 M	130 M	124 M	1 M	2 M
Total weights	60 k	61 M	146 M	138 M	7 M	25.5 M
Total MACs	341 k	724 M	2.8 G	15.5 G	1.43 G	3.9 G
Pretrained model website	[77]‡	[78, 79]	n/a	[78, 79, 80]	[78, 79, 80]	[78, 79, 80]

(i.e., the CNN is run only once), which is more consistent with what would likely be deployed in real-time and/or energy-constrained applications.

LeNet [20] 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 followed by two FC layers [71]. 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 [7] was the first CNN to win the ImageNet Challenge in 2012. It consists of five CONV layers followed by 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 three channels of the filter correspond to the red, green, and blue components of the input image. A ReLU nonlinearity 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. This approach is referred to as "grouped convolution" and illustrated in Figure 2.8.[14] 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 [72] has a very similar architecture to AlexNet with five CONV layers followed by 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 FC 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 [73] goes deeper to 16 layers consisting of 13 CONV layers followed by 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 effective receptive fields, as shown in Figure 2.9a. 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.

GoogLeNet [74] goes even deeper with 22 layers. It introduced an inception module, shown in Figure 2.10, whose input is distributed through multiple feed-forward connections to several parallel layers. These parallel layers contain different sized filters (i.e., 1×1, 3×3, 5×5), along with 3×3 max-pooling, 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 train-

[14]This grouped convolution approach is applied more aggressively when performing co-design of algorithms and hardware to reduce complexity, which will be discussed in Chapter 9.

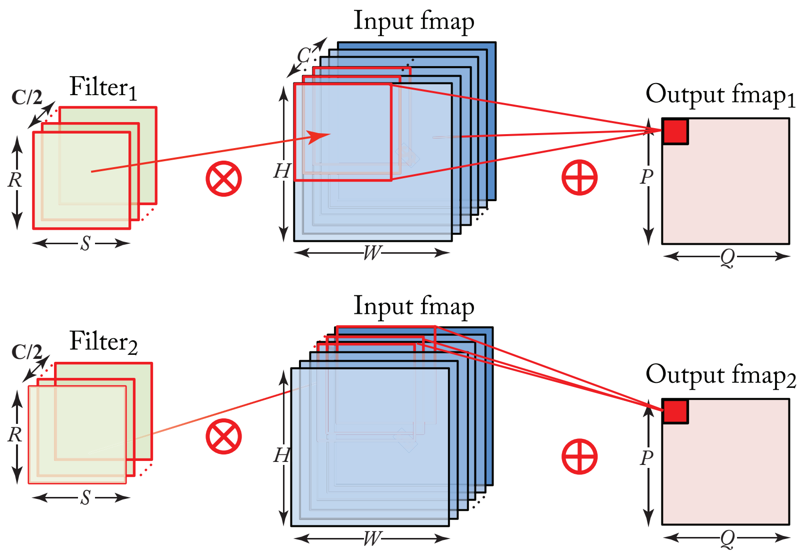
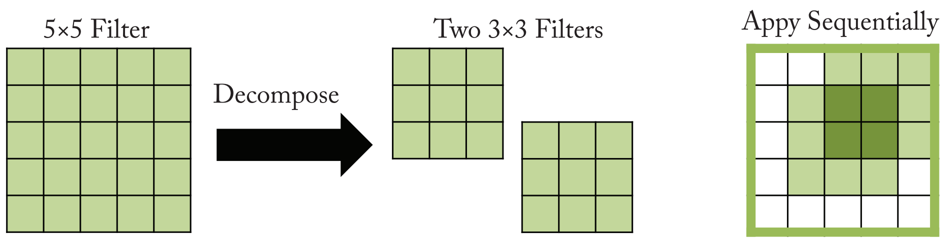
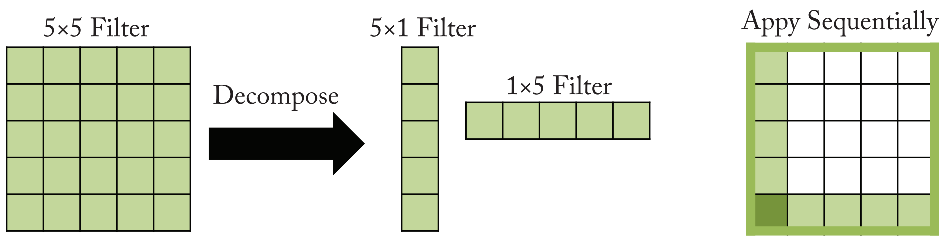


Figure 2.8: An example of dividing feature map into two *grouped convolutions*. Each filter requires 2× fewer weights and multiplications.



(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.

Figure 2.9: Decomposing larger filters into smaller filters.

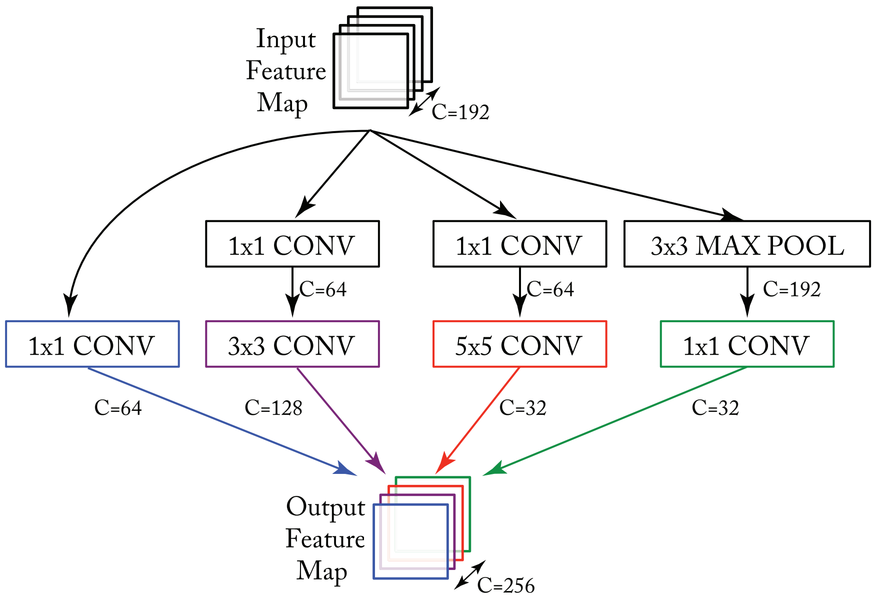


Figure 2.10: Inception module from GoogLeNet [74] with example channel lengths. Note that each CONV layer is followed by a ReLU (not drawn).

ing 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 [75], as shown in Figure 2.11. The 22 layers consist of three CONV layers, followed by nine inceptions modules (each of which are two CONV layers deep), and one FC layer. The number of FC layers was reduce from three to one using a global average pooling layer, which summarizes the large feature map from the CONV layers into one value; global pooling will be discussed in more detail in Section 9.1.2. Since its introduction in 2014, GoogLeNet (also referred to as Inception) has multiple versions: v1 (described here), v3,[15] and v4. Inception-v3 decomposes the convolutions by using smaller 1-D filters, as shown in Figure 2.9b, to reduce number of MACs and weights in order to go deeper to 42 layers. In conjunction with batch normalization [66], v3 achieves over 3% lower Top-5 error than v1 with 2.5× more MACs [76]. Inception-v4 uses residual connections [77], described in the next section, for a 0.4% reduction in error.

ResNet [24], also known as Residual Net, uses feed-forward connections that connects to layers beyond the immediate next layer (often referred to as *residual*, *skip* or *identity* connections); these connections enable a DNN with many layers (e.g., 34 or more) to be trainable. It was

[15] v2 is very similar to v3.

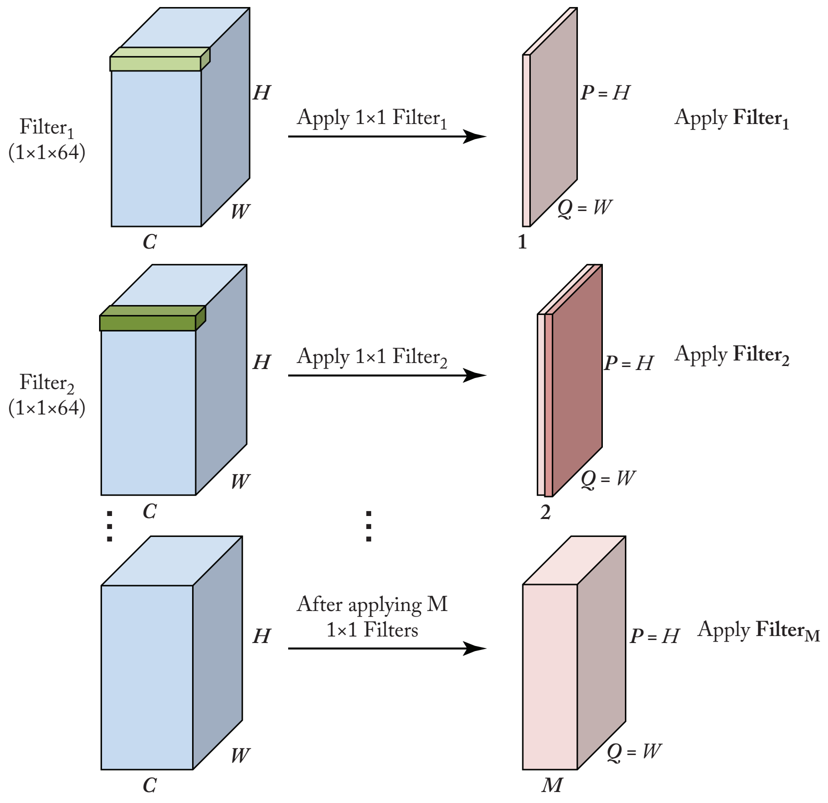
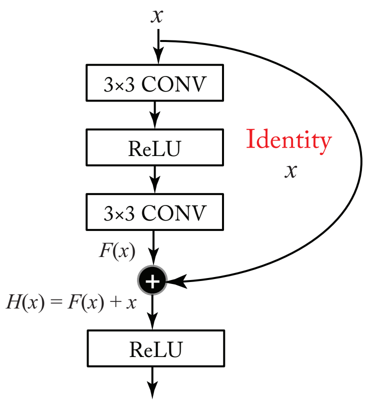
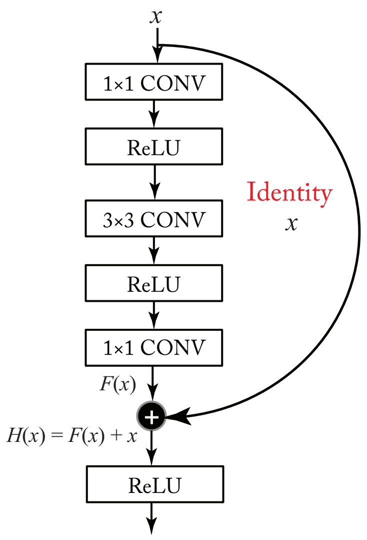


Figure 2.11: Apply 1×1×C filter (usually referred to as 1×1) to capture cross-channel correlation, but no spatial correlation. This bottleneck approach reduces the number of channels in next layer assuming the number of filters applied (M) is less than the original number of channels (C).

the first entry CNN 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 [78]; 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. ResNet introduces a "shortcut" module which contains an identity connection such that the weight layers (i.e., CONV layers) can be skipped, as shown in Figure 2.12. 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. ResNet also uses the "bottleneck" approach of using 1×1 filters to reduce the number of weights. As a result, the two layers in the shortcut module are replaced by three layers (1×1, 3×3, 1×1) where the first 1×1 layer reduces the number of activations and thus weights in the 3×3 layer, the last 1×1 layer restores



(a) Without bottleneck



(a) With bottleneck

Figure 2.12: Shortcut module from ResNet [24]. Note that ReLU following last CONV layer in shortcut is *after* the addition.

the number of activations in the output of the third layer. ResNet-50 consists of one CONV layer, followed by 16 shortcut layers (each of which are 3 CONV layers deep), and 1 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 CNNs shown in Table 2.2. 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 nonlinear functions that are more discriminative and also provides more levels of hierarchy in the learned representation [24, 73, 74, 79]. 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 targeted at CNNs should be on addressing the efficiency of the CONV layers, which in many domains are increasingly important.

Since ResNet, there have been several other notable networks that have been proposed to increase accuracy. *DenseNet* [84] extends the concept of skip connections by adding skip con-

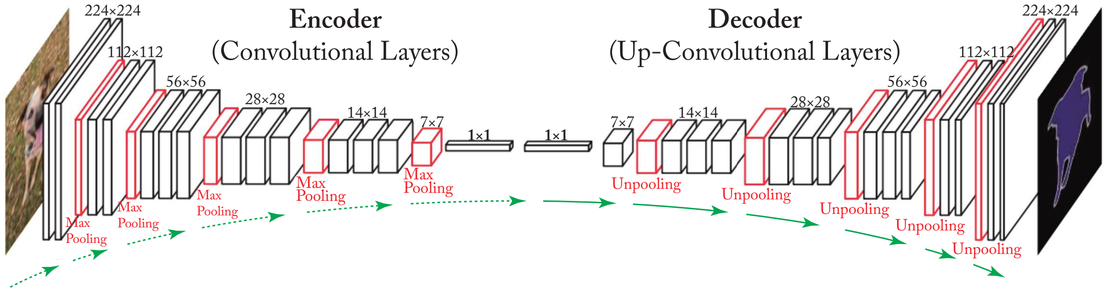


Figure 2.13: Auto Encoder network for semantic segmentation. Feature maps along with pooling and upsampling layers are shown. (Figure adapted from [92].)

nection from *multiple* previous layers to strengthen feature map propagation and feature reuse. This concept, commonly referred to as *feature aggregation*, continues to be widely explored. *WideNet* [85] proposes increasing the width (i.e., the number of filters) rather than depth of network, which has the added benefit that increasing width is more parallel-friendly than increasing depth. *ResNeXt* [86] proposes increasing the number of convolution groups (referred to as cardinality) instead of depth and width of network and was used as part of the winning entry for ImageNet in 2017. Finally, *EfficientNet* [87] proposes uniformly scaling all dimensions including depth, width, and resolution rather than focusing on a single dimension since there is an interplay between the different dimensions (e.g., to support higher input image resolution, the DNN needs higher depth to increase the receptive field and higher width to capture more fine-grained patterns). WideNet, ResNeXt, and EfficientNet demonstrate that there exists methods beyond increasing depth for increasing accuracy, and thus highlights that there remains much to be explored and understood about the relationship between layer shape, number of layers, and accuracy.

2.5 OTHER DNNs

There are other types of DNNs beyond CNNs including Recurrent Neural Networks (RNNs) [88, 89], Transformers [68], Auto Encoders (AEs) [90], and General Adversarial Networks (GANs) [91]. The diverse types of DNNs allow them to handle a wide range of inputs for a wide range of tasks. For instance, RNNs and Transformers are often used to handle sequential data that can have variable length (e.g., audio for speech recognition, or text for natural language processing). AEs and GANs can be used to generate dense output predictions by combining encoder and decoder networks. Example applications that use AEs include predicting pixel-wise depth values for depth estimation [64] and assigning pixel-wise class labels for semantic segmentation [92], as shown in Figure 2.13. Example applications that use GANs to generate images with the same statistics as the training set include image synthesis [93] and style transfer [94].

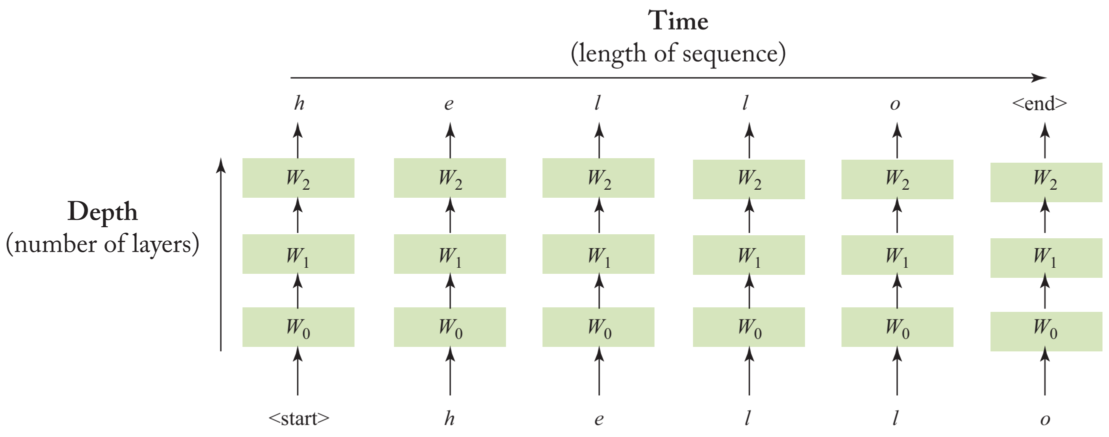


Figure 2.14: Dependencies in RNN are in both the time and depth dimension. The same weights (W_i) are used across time, while different weights are used across depth. (Figure adapted from [4].)

While their applications may differ from the CNNs described in Section 2.4, many of the building blocks and primitive layers are similar. For instance, RNNs and transformers heavily rely on matrix multiplications, which means that they have similar challenges as FC layers (e.g., they are memory bound due to lack of data reuse); thus, many of the techniques used to accelerate FC layers can also be used to accelerate RNNs and transformers (e.g., tiling discussed in Chapter 4, network pruning discussed in Chapter 8, etc.). Similarly, the decoder network of GANs and AEs for image processing use *up-convolution layers*, which involves upsampling the input feature map using zero insertion (unpooling) before applying a convolution; thus, many of the techniques used to accelerate CONV layers can also be used to accelerate the decoder network of GANs and AEs for image processing (e.g., exploit input activation sparsity discussed in Chapter 8).

While the dominant compute aspect of these DNNs are similar to CNNs, they do often require some other forms of compute. For instance, RNNs, particularly Long Short-Term Memory networks (LSTMs) [95], require support of element-wise multiplications as well a variety of nonlinear functions (sigmoid, tanh), unlike CNNs which typically only use ReLU. However, these operations do not tend to dominate run-time or energy consumption; they can be computed in software [96] or the nonlinear functions can be approximated by piecewise linear look up tables [97]. For GANs and AEs, additional support is required for upsampling.

Finally, RNNs have additional dependencies since the output of a layer is fed back to its input, as shown in Figure 2.14. For instance, the inputs to layer i at time t depends on the output of layer $i-1$ at time t and layer i at time $t-1$. This is similar to the dependency across layers, in that the output of layer i is the input to layer $i+1$. These dependencies limit what inputs can be processed in parallel (e.g., within the same batch). For DNNs with feed-forward