MORGAN & CLAYPOOL PUBLISHERS

Deep Learning Systems

Algorithms, Compilers, and Processors for Large-Scale Production

Andres Rodriguez

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

Natalie Enright Jerger, *Series Editor*

Deep Learning Systems

Algorithms, Compilers, and Processors for Large-Scale Production

Deep Learning Systems: Algorithms, Compilers, and Processors for Large-Scale Production

Andres Rodriguez

www.morganclaypool.com

ISBN: 9781681739663 paperback
ISBN: 9781681739670 ebook
ISBN: 9781681739687 hardcover

DOI 10.2200/S01046ED1V01Y202009CAC053

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

Lecture #53
Editor: Natalie Enright Jerger, *University of Toronto*
Editor Emerita: Margaret Martonosi, *Princeton University*
Founding Editor Emeritus: Mark D. Hill, *University of Wisconsin, Madison*
Series ISSN
Print 1935-3235 Electronic 1935-3243

Deep Learning Systems

Algorithms, Compilers, and Processors for Large-Scale Production

Andres Rodriguez
Intel

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE #53

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

This book describes deep learning systems: the algorithms, compilers, and processor components to efficiently train and deploy deep learning models for commercial applications.

The exponential growth in computational power is slowing at a time when the amount of compute consumed by state-of-the-art deep learning (DL) workloads is rapidly growing. Model size, serving latency, and power constraints are a significant challenge in the deployment of DL models for many applications. Therefore, it is imperative to codesign algorithms, compilers, and hardware to accelerate advances in this field with holistic system-level and algorithm solutions that improve performance, power, and efficiency.

Advancing DL systems generally involves three types of engineers: (1) data scientists that utilize and develop DL algorithms in partnership with domain experts, such as medical, economic, or climate scientists; (2) hardware designers that develop specialized hardware to accelerate the components in the DL models; and (3) performance and compiler engineers that optimize software to run more efficiently on a given hardware. Hardware engineers should be aware of the characteristics and components of production and academic models likely to be adopted by industry to guide design decisions impacting future hardware. Data scientists should be aware of deployment platform constraints when designing models. Performance engineers should support optimizations across diverse models, libraries, and hardware targets.

The purpose of this book is to provide a solid understanding of (1) the design, training, and applications of DL algorithms in industry; (2) the compiler techniques to map deep learning code to hardware targets; and (3) the critical hardware features that accelerate DL systems. This book aims to facilitate co-innovation for the advancement of DL systems. It is written for engineers working in one or more of these areas who seek to understand the entire system stack in order to better collaborate with engineers working in other parts of the system stack.

The book details advancements and adoption of DL models in industry, explains the training and deployment process, describes the essential hardware architectural features needed for today's and future models, and details advances in DL compilers to efficiently execute algorithms across various hardware targets.

Unique in this book is the holistic exposition of the entire DL system stack, the emphasis on commercial applications, and the practical techniques to design models and accelerate their performance. The author is fortunate to work with hardware, software, data scientist, and research teams across many high-technology companies with hyperscale data centers. These companies employ many of the examples and methods provided throughout the book.

KEYWORDS

deep learning, machine learning, artificial intelligence, distributed training systems, inference, accelerators, processors, architectures, compilers, optimizations

To Isabel, Tomas, and David

Contents

Preface

Many concepts throughout the book are interdependent and often introduced iteratively with a reference to the section covering the concept in detail. If you are new to this field, read the introductory chapter in its entirety and each chapter's introductory section and concluding paragraph to capture some of the key takeaways. Then, go back and read each chapter in its entirety. A background in linear algebra, calculus, programming, compilers, and computer architecture may be helpful for some parts but not required. The book is organized as follow:

Chapter 1 starts with an introduction to essential concepts detailed throughout the book. We review the history and applications of deep learning (DL). We discuss various types of topologies employed in industry and academia across multiple domains. We also provide an example of training a simple DL model and introduce some of the architectural design considerations.

Chapter 2 covers the building blocks of models used in production. We describe which of these building blocks are compute bound and which are memory bandwidth bound.

Chapter 3 covers the applications benefiting the most from DL, the prevalent models employed in industry, as well as academic trends likely to be adopted commercially over the next few years. We review recommender system, computer vision, natural language processing (NLP), and reinforcement learning (RL) models.

Chapter 4 covers the training process domain experts should follow to adopt DL algorithms successfully. We review topology design considerations employed by data scientists, such as weight initialization, objective functions, optimization algorithms, training with a limited dataset, dealing with data imbalances, and training with limited memory. We also describe the mathematical details behind the backpropagation algorithm to train models.

Chapter 5 covers distributed algorithms adopted in data centers and edge devices (known as federated learning). We discuss the progress and challenges with data and model parallelism. We also review communication primitives and AllReduce algorithms.

Chapter 6 covers the lower numerical formats used in production and academia. These formats can provide computational performance advantages over the standard 32-bit single-precision floating-point, sometimes at the expense of lower statistical performance (accuracy). We also discuss pruning and compression techniques that further reduce the memory footprint.

Chapter 7 covers hardware architectural designs. We review the basics of computer architecture, reasons for the slower growth in computational power, and ways to partially mitigate this slowdown. We explain the roofline model and the important hardware characteristics for serving and multinode training. We also discuss CPUs, GPUs, CRGAs, FPGAs, DSPs, and

ASICs, their advantages and disadvantages, and the prominent DL processors and platforms available in the market or in development.

Chapter 8 covers high-level languages and compilers. We review language types and explain the basics of the compilation process. We discuss front-end compilers that transform a program to an LLVM internal representation (IR) and the LLVM back-end compiler. We also describe the standard compiler optimizations passes for DL workloads.

Chapter 9 covers the frameworks and DL compilers. We review in detail the TensorFlow and PyTorch frameworks and discuss various DL compilers in development.

Chapter 10 concludes with a look at future opportunities and challenges. We discuss the opportunities to use machine learning algorithms to advance various parts of the DL system stack. We discuss some challenges, such as security, interpretability, and the social impact of these systems. We also offer some concluding remarks.

Acknowledgments

The task of writing this book would not have been possible without the support of many individuals. At the risk of unintentionally leaving some out, I want to thank the following for their guidance, time, and help discussing and improving the content.

Thanks to Michael Morgan at Morgan & Claypool for inviting me to write this book as well as Natalie Enright Jerger, Christine Kiilerich, and C.L. Tondo for their continuous encouragement and excellent support throughout the writing and publishing process. Thanks to Parthasarathy Ranganathan for his guidance and for introducing me to Michael and Natalie.

Thanks to Cliff Young, Natalie Enright Jerger, Nikhil Murthy, Nicholas Lee, and Joanne Yuan for providing detailed comments that were invaluable in improving the entire manuscript.

Thanks to Robert Zak, Wolff Daniel Dobson, Vinay Phegade, Roman Dubtsov, Kreig DuBose, Matthew Brookhart, Chinnikrishna Kothapalli, Steven (Cap) Rogers, Dheevatsa Mudigere, Carlos Escapa, Tatiana Shpeisman, Vijayakumar (Kumar) Bhagavatula, Mourad Gouicem, AG Ramesh, Jian Hui Li, Romir Desai, Vadim Pirogov, Michael Goldsmith, Chris Browning, Gal Novik, Sherine Abdelhak, Abigail Wen, Md Faijul Amin, Tristan Webb, Marcel Nassar, Koichi Yamada, Anna Bethke, Ivan Kuzmin, Anahita Bhiwandiwalla, Mariano Phelipp, Brian Retford, Tiffany Shih, Jayaram Bobba, Edward Groden, Anthony Reina, Bin Wei, Jacek Czaja, Etay Meiri, Luke Ge, Ran Cohen, Derek Osborne, Jayarama Shenoy, Michael Greenfield, Madhusudhan Rangarajan, Eric Lin, Lei Xia, Albert Hu, Brinda Ganesh, Diego Caballero, Tatyana Primak, Naveen Mellempudi, Mohammad Ashraf Bhuiyan, Rajesh Poornachandran, Rinat Rappoport, Xin Wang, Yoann Foucher, Pujiang He, Jun Jin, Eric Gardner, Adam Straw, Scott Cyphers, Brian Golembiewski, Clayne Robison, Sangeeta Bhattacharya, Ravi Panchumarthy, Patric Zhao, Derssie Mebratu, Anisha Gartia, Shamima Najnin, Rajendrakumar Chinnaiyan, Zhenlin Luo, Chris Banyai, Vinod Devarampati, Alex Heinecke, Evarist Fomenko, Milind Pandit, Lei Shao, Yong Wu, Sameh Gobriel, Andrey Nikolaev, Nawab Ali, Bernhard Friebe, Nikhil Deshpande, Shashikant Kale, Amir Gholaminejad, Indu Kalyanaraman, and Greg Leeming for their excellent comments on portions of the book relevant to their respective expertise, the discussions to ensure correctness, or proofreading early drafts to improve clarity. Thanks to Roberto Gauna and Travis Belnap for their assistance making and improving many of the figures.

My biggest thanks goes to Mary-Kathryn for her unwavering friendship and fervent support, which made the writing of this book possible. And my second biggest thanks to our children Isabel, Tomas, and David, who inspire me, and who were patient and forgiving when Papi was busy writing and could not play. This book is dedicated to you. May the progress in artificial intelligence and deep learning systems make the world for you and your generation a better place.

Disclaimer: The views expressed in the book are my own and do not represent those of the Intel Corporation, previous employers, or those acknowledged. Details regarding software and hardware products come from publicly disclosed information, which may not represent the latest status of those products.

Errata: Please submit your comments and errata to `deep.learning.systems@gmail.com`. Thanks in advance for taking the time to contribute.

Andres Rodriguez
October 2020

CHAPTER 1 Introduction

A deep learning (DL) model is a function that maps input data to an output prediction. To improve the accuracy of the prediction in complex tasks, DL models are increasingly requiring more compute, memory, bandwidth, and power, particularly during training. The number of computations required to train and deploy state-of-the-art models doubles every ~3.4 months [DH18]. The required computation scales at least as a fourth-order polynomial with respect to the accuracy and, for some tasks, as a nineth-order polynomial [TGL+20]. This appetite for more compute far outstrips the compute growth trajectory in hardware and is unsustainable. In addition, the main memory bandwidth is becoming a more significant bottleneck; computational capacity is growing much faster than memory bandwidth, and many algorithms are already bandwidth bound.

The evolution of computational growth is driving innovations in DL architectures. Improvements in transistor design and manufacturing no longer result in the previously biennial 2× general-purpose computational growth. The amount of dark silicon, where transistors cannot operate at the nominal voltage, is increasing. This motivates the exploitation of transistors for domain-specific circuitry.

Data scientists, optimization (performance) engineers, and hardware architects must collaborate on designing DL systems to continue the current pace of innovation. They need to be aware of the algorithmic trends and design DL systems with a 3–5 year horizon. These designs should balance general-purpose and domain-specific computing and accommodate for unknown future models.

The characteristics of DL systems vary widely depending on the end-user and operating environment. Researchers experimenting with a broad spectrum of new topologies (also known as DL algorithms or neural networks) require higher flexibility and programmability than engineers training and deploying established topologies. Furthermore, even established topologies have vastly different computational profiles. For instance, an image classification model may have a compute-to-data ratio three orders of magnitude higher than that of a language translation model.

A mixture of specialized hardware, higher bandwidth, compression, sparsity, smaller numerical representations, multichip communication, and other innovations is required to satisfy the appetite for DL compute. Each 2× in performance gain requires new hardware, compiler, and algorithmic co-innovations.

Advances in software compilers are critical to support the Cambrian explosion in DL hardware and to effectively compile models to different hardware targets. Compilers are essential to mitigate the cost of evaluating or adopting various hardware designs. A good compiler generates code that runs efficiently and speedily executes. That is, the generated code takes advantage of the computational capacity and memory hierarchy of the hardware so the compute units have high utilization. Several efforts, detailed in Chapter 9, are ongoing toward making this possible.

The purposes of this book are (1) to provide a solid understanding of the design, training, and applications of DL algorithms, the compiler techniques, and the critical processor features to accelerate DL systems, and (2) to facilitate co-innovation and advancement of DL systems.

In this chapter, we introduce the fundamental concepts detailed throughout the book. We review the history, applications, and types of DL algorithms. We provide an example of training a simple model and introduce some of the architectural design considerations. We also introduce the mathematical notation used throughout parts of the books.

1.1 DEEP LEARNING IN ACTION

DL models are tightly integrated into various areas of modern society. Recommender models recommend ads to click, products to buy, movies to watch, social contacts to add, and news and social posts to read. Language models facilitate interactions between people who speak different languages. Speech recognition and speech generation advance human-machine interactions in automated assistants. Ranking models improve search engine results. Sequence models enhance route planning in navigation systems. Visual models detect persons, actions, and malignant cells in MRI and X-ray films.

Other DL applications are drug discovery, Alzheimer diagnosis prediction, asteroid identification, GUI testing, fraud detection, trading and other financial applications, neutrino detection, robotics, music and art generation, gaming, circuit design, code compilation, HPC system failure detection, and many more.

Despite their tremendous success across multiple prediction domains, DL algorithms have limitations. They are not yet reliable in some behavior prediction, such as identifying recidivism, job success, terrorist risk, and at-risk kids [Nar19]. Other areas with limited functionality are personalized assistants and chatbots.

Another limitation is in Artificial General Intelligence (AGI), sometimes referred to as Strong AI. AGI is where machines exhibit human intelligence traits, such as consciousness and self-awareness. The tentative time when machines reach this capability was coined by John Von Neumann as the singularity. There are mixed opinions in the AI community on the timing of singularity ranging from later in this century to never. Given the extremely speculative nature, AGI is not discussed further.

The adoption of DL is still in its infancy. There are simpler machine learning algorithms that require less data and compute, which are broadly adopted across industries to analyze data and make predictions. These include linear regression, logistic regression, XGBoost, and LightGBM (do not worry if you are unfamiliar with these algorithms). The majority of the winning solutions to popular Kaggle challenges involve these computationally simpler algorithms.

Nevertheless, interest in DL is extraordinarily high, and its adoption is rapidly growing. High-technology companies with warehouse-scale computers (WSC) [BHR18], referred to hereafter as hyperscale companies or hyperscalers, use DL in production primarily for these workloads (in order of importance):

1. Recommendations (due to the monetization benefits) for personalized ads, social media content, and product recommendations.

2. Natural language processing (NLP) for human-machine interaction by virtual assistants (Alexa, Siri, Cortana, G-Assistant, and Duer) and chatbots/service-bots, to combat language toxicity, for language translation, and as a preprocessing step to a recommendation workload.

3. Computer vision for biometrics, autonomous driving, image colorization, medical diagnosis, and art generation.

Recommender topologies are critical to several hyperscalers; they are more closely tied to revenue generation than computer vision and NLP topologies. The overall number of servers in data centers dedicated to recommenders is likely higher than NLP and computer vision. For instance, at Facebook, recommender models account for over 50% of all training cycles and over 80% of all their inference cycles [Haz20, NKM+20].

Computer vision topologies are widely adopted across enterprise data centers and on client devices, such as mobile phones. When companies begin the adoption of DL, they often start with computer vision topologies. These topologies are the most matured and provide significant gains over non-DL approaches. Given that several open-source datasets are available in this area, the overwhelming majority of academic papers focus in computer vision: 82%; compared to 16% for NLP and 2% for recommenders due to limited public datasets [Haz20].

Model training and serving have different requirements. Training can be computationally intensive. For instance, the popular image classification ResNet-50 model requires about 1 Exa (10^{18}) operations and is considered small by today's standards [YZH+18]. Training the much larger Megatron-LM model requires 12 Zetta (12×10^{21}) operations [SPP+19]. Other models, such as some recommenders, have unique challenges often not only requiring high compute but large memory capacity and high network and memory bandwidth.

During the training process, multiple samples are processed in parallel, improving data reuse and hardware utilization. Except for memory capacity bounded workloads, most large model training happens on GPUs due to their higher (compared to CPUs) total operations per second, higher memory bandwidth, and software ecosystem.

Serving, also known as inference, prediction, deployment, testing, or scoring, is usually part of a broader application. While one inference cycle requires little compute compared to

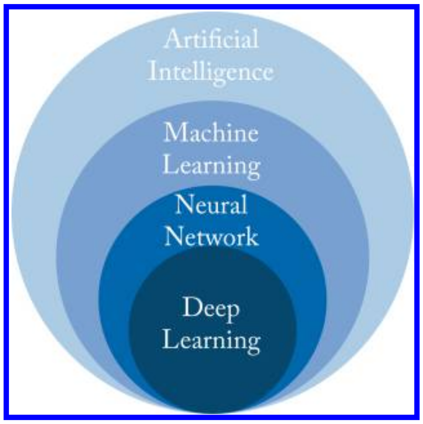


Figure 1.1: Deep learning is a subset of neural networks, which is a subset of machine learning, which is a subset of artificial intelligence.

training, the total compute spent on inference on a given model dwarfs that of training over the entire life span of the model.

Serving is typically latency bounded. Product recommendations, search results, voice assistant queries, and pedestrian detection in autonomous vehicles require results within a prespecified latency constraint. Thus, during serving, only one or a few samples are typically processed to meet the latency requirement. Effectively parallelizing the serving computations for one data sample across a large number of cores is challenging. For this reason, GPUs (and CPUs to a lesser extend) suffer from poor compute utilization during serving. There is an opportunity for hardware architects to design better low-latency processors and minimize idle compute cycles, detailed in Chapter 7.

Serving in data centers typically happens on CPUs due to their higher availability, higher core frequency, and higher compute utilization for small batches. Given the parallelization challenges when using one data sample, fewer faster cores in a CPU may be advantageous over many slower cores in a GPU. Using more cores can further reduce the latency at the expense of lower core utilization (due to the core-to-core communication overhead). However, as models grow and require more compute, some companies are transitioning to GPUs or experimenting with dedicated processors for inference. In addition, low power (smaller) GPUs or GPUs with virtualization reduces the number of cores allocated to a workload, which improves core utilization.

1.2 AI, ML, NN, AND DL

The terms artificial intelligence (AI), machine learning (ML), neural network (NN), and deep learning (DL) are often used interchangeably. While there are no agreed-upon standard definitions, the following are common and captured in Figure 1.1.

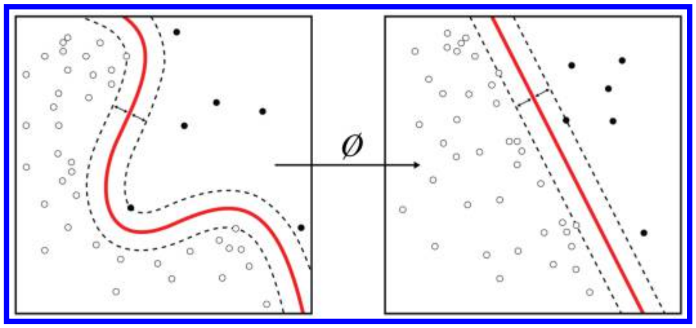


Figure 1.2: A machine learning algorithm maps the input data to a space or manifold where the data can be classified with a linear classifier. Source: [Wik11] (CC BY-SA 4.0).

AI is any program or system that can learn, act, or adapt. The recent popularity of AI comes from advances in ML algorithms, specifically in DL. An ML model is a program that learns a function that maps the input data (or features extracted from the input data) to a desired output. Geometrically, this mapping is from a vector space where the data is not linearly separable to a vector space where the data is linearly separable, as illustrated in Figure 1.2. These vector spaces are formally known as Hilbert spaces or manifolds. The mapping function or statistical performance (accuracy) of the model usually improves with more data.

NN models, also called artificial neural networks (ANNs), are typically composed of simple nonlinear functions, called layers, stacked together to represent complex functions or mappings. Stacking multiple linear functions results in one linear function that can be represented with one layer, and would negate the benefit of multilayer mappings. Thus, the need for nonlinear functions. DL models, sometimes called deep neural networks (DNNs), are NN models with more than three layers and are end-to-end differentiable. Traditional machine learning (non-NN ML) models and NN models with 1–3 layers are also called shallow models.

A difference between traditional ML and most of DL is traditional ML relies on domain experts to specify key features to use for the given task. In contrast, DL typically learns these features at the expense of requiring more data and compute. For decades, computer vision experts spent significant efforts studying image features to improve detection [FGM+10]. DL practitioners with limited computer vision expertise demonstrated that NNs were able to learn features with increasing complexity at each layer and outperform state-of-the-art techniques in image detection and classification tasks [KSH12].

DL models are particularly advantageous, although requiring much more data and compute, over traditional ML models for workloads where the relationship between features cannot be reasonably approximated, such as with visual, text, or speech data. Traditional ML models continue to be popular with tabular or structured data where the feature relationships can be approximated, for instance, using a Bayesian model to encode the hierarchical relationships manually (do not worry if you are unfamiliar with Bayesian models) [DWO+19].

1.3 BRIEF HISTORY OF NEURAL NETWORKS

NNs were popularized in the 1960s and used for binary classification. Their popularity diminished in the 1970s when NNs did not deliver on the hype. Interest in NNs increased in the mid-1980s when the backpropagation algorithm (detailed in Chapter 4) was rediscovered, facilitating the training of multilayer NNs to learn more complex classifiers. In the mid-1990s, most of the AI focus shifted toward support vector machines (SVMs), a class of ML algorithms with theoretical performance bounds. The NN community refers to the 1970s as the first AI winter and the mid-1990s to early 2000s as the second AI winter due to the limited funding of and progress in NNs (these should be called NN winters since AI is bigger than NNs).

During the past decade, there has been a revived interest as NN have vastly outperformed other techniques, particularly for vision and natural language processing tasks. This recent success is due to faster and cheaper hardware, more massive datasets, improved algorithms, and open-source software [SAD+20]. Researchers from competing companies often publish their algorithms and training methodologies (but typically not their trained models or datasets); thus, they build on each other's knowledge and accelerate progress.

1.4 TYPES OF LEARNING

ML algorithms usually fall into one of four learning types or categories: supervised, unsupervised, semi-supervised, and reinforcement learning (RL), as shown in Figure 1.3 and discussed below. Despite the names, all these learnings are "supervised" in that they required a human to explicitly define the cost function that determines what is good or bad. Note that a different way to categorize ML algorithms is as discriminative or generative. A discriminative algorithm learns to map the input data to a probability distribution. A generative algorithm learns statistical properties about the data and generates new data.

1.4.1 SUPERVISED LEARNING

Supervised learning is the most common type used in industry due to the monetization advantages and it is the primary, but not exclusive, focus of the models presented in this book. Supervised learning uses annotated or labeled data for training, meaning the ground truth or the desired output of the model for each data sample in the training dataset is known. Training involves learning a function that approximately maps the input to the desired output. The function can be a regression or a classification function. Regression functions have a numerical or continuous output, such as the price of a house (the input data would be features of the house, such as house size and local school rating). Classification functions have discrete or categorical outputs, such as {car, pedestrian, road} (the input data would be image pixels). The majority of DL models used in industry are for classification tasks. Figure 1.3a shows a classification example with the learned linear decision boundaries between three different classes. The green circles, red crosses, and blue stars represent 2D features extracted from samples in each class.

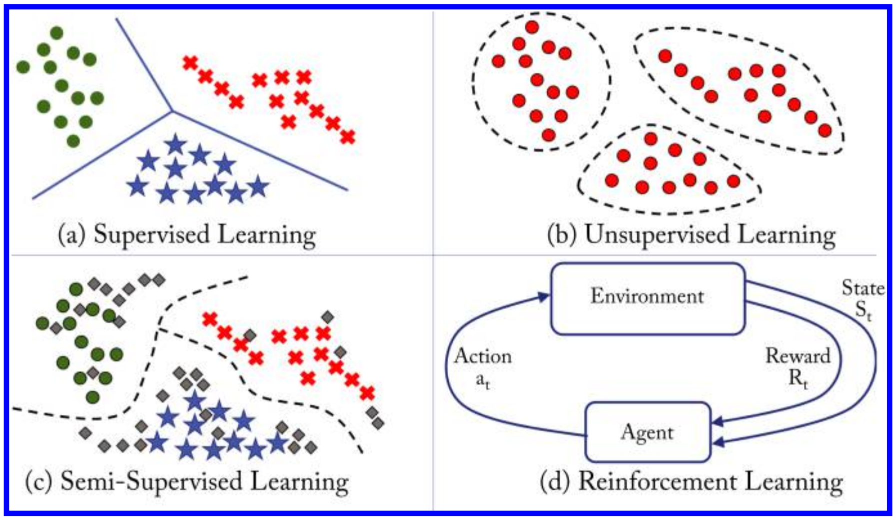

(a) Supervised Learning
(b) Unsupervised Learning
(c) Semi-Supervised Learning
Environment
State S_t
Action a_t
Reward R_t
Agent
(d) Reinforcement Learning

Figure 1.3: The four types of ML algorithms.

Examples of supervised learning tasks with input data and labels are (task: input data → label):

- Image classification: pixels → class label of the object in an image
- Image detection: pixels → bounding box around each object in an image and the class label of those objects
- Recommender system: shopping history, IP address, products → product purchased
- Machine translation: sentence in the source language → sentence in the target language
- Speech recognition: sound waves → written text
- Speech generation or text-to-speech (TTS): written text → sound waves
- Regression analysis: house size, local school rating → price of the house

1.4.2 UNSUPERVISED AND SELF-SUPERVISED LEARNING

Unsupervised learning learns patterns in unlabeled data. Figure 1.3b shows a clustering example with the learned clusters on unlabeled data. Self-supervised learning is a subset of unsupervised learning and includes learning embeddings and predicting missing words or pixels in text or images. For instance, each word in a 10,000-words-language can be represented as a 10,000-dimensional vector of all zeros except for a one at the index of the particular word. This vector is called a one-hot vector, shown in Figure 1.4. Self-supervised learning models can learn to map

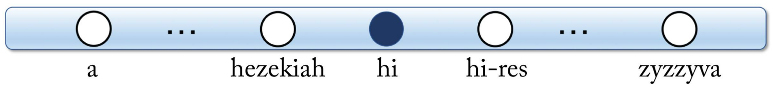


Figure 1.4: One-hot vector. All entries are zero except a one at the vector entry corresponding to the word.

this sparse vector to a small and dense vector representation. Other examples are learning dense vector representations for persons in a social network and products in a large catalog. These dense vector representations are often the inputs into a supervised learning model.

1.4.3 SEMI-SUPERVISED LEARNING

Semi-supervised learning combines techniques from supervised and unsupervised learning. Figure 1.3c shows a small labeled dataset augmented with a much larger unlabeled dataset to improve (over the supervised learning algorithm) the decision boundaries between the classes. While most of the past decade's success has been with supervised learning, semi-supervised learning is a promising approach given the massive amounts of unlabeled data generated each day. Moreover, to draw inspiration from human learning, children appear to learn using mostly unlabeled data. However, adoption in industry is limited.

1.4.4 REINFORCEMENT LEARNING

RL is used to teach an agent to perform certain actions based on rewards received after a set of actions. The agent's goal is to maximize the rewards. Figure 1.3d depicts an agent interacting with the environment. The agent gets a reward based on the outcome of a given action. There are three types of RL algorithms: Q-learning, policy optimization and model-based, detailed in Section 3.4.

1.5 TYPES OF TOPOLOGIES

A *topology* is a computation graph that represents the structure or architecture of a NN, as shown in Figure 1.5. The nodes represent operations on tensors (multidimensional arrays), and the edges dictate the data flow and data-dependencies between nodes.

The main types of topologies used in commercial applications are multilayer perceptrons (MLPs), convolution neural networks (CNNs), recurrent neural networks (RNNs), and transformer networks. These topologies are introduced below and detailed in Chapter 3. Other types of topologies common in research with some adoption in commercial applications are graph neural networks (GNNs), adversarial networks (ANs), and autoencoders (AEs). Bayesian neural networks (BNNs) and spiking neural networks (SNNs) are limited to research.

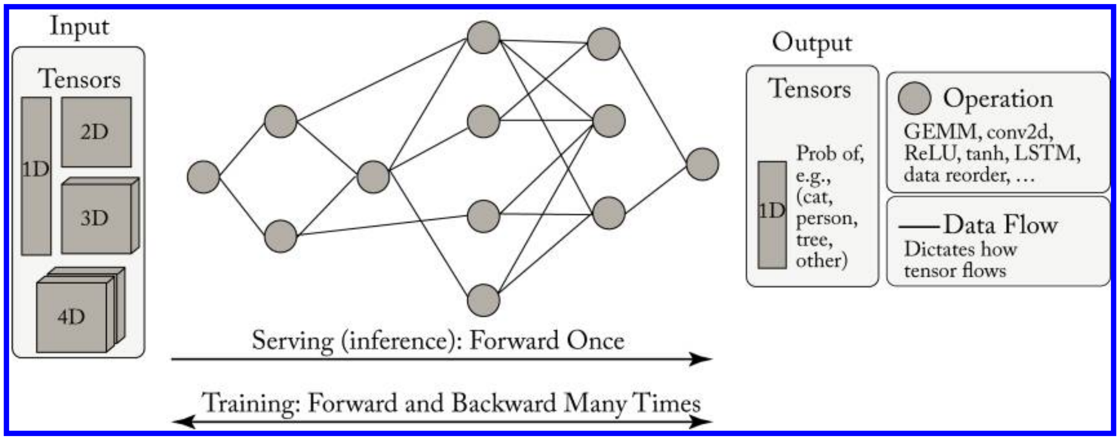

Input
Tensors
1D
2D
3D
4D
Serving (inference): Forward Once
Training: Forward and Backward Many Times
Output
Tensors
1D
Prob of, e.g., (cat, person, tree, other)
Operation
GEMM, conv2d, ReLU, tanh, LSTM, data reorder, …
Data Flow
Dictates how tensor flows

Figure 1.5: A computation graph takes a tensor input and produces a tensor output. Serving involves typically one forward propagation. Training involves numerous forward and backward iteration cycles.

1.5.1 MULTILAYER PERCEPTRON

A feedforward neural network (FFNN) is a directed acyclic graph (DAG) with an input layer, an output layer, and one or more layers in between called hidden layers. The nodes in the input layer have no parent nodes, and the nodes in the output layer have no children nodes. The inputs are either learned or extracted from the data. These models are the most widely used by hyperscalers, in particular (but not exclusively), for recommender systems.

An MLP is a vanilla FFNN with affine layers, also called fully connected layers, with each layer followed by an activation function. An affine layer is composed of units that linearly combine the weighted outputs or activations from the previous layer plus a bias (the bias is considered another weight). Using multiple layers enables the MLP model to represent complex nonlinear functions [HSW89]. Geometrically, an MLP model attempts to map one vector space to another vector space, where the data is linearly separable, via multiple nonlinear transformations, as shown in Figure 1.2. In this new manifold, the last FFNN layer is a linear classifier that separates most of the data from different classes.

Figure 1.6 shows a four-layer MLP used for digit classification with Layer 0 having $D^{(0)} = 784$ units corresponding to each pixel (the input image has 28×28 pixels), Layers 1 and 2 are hidden units having $D^{(1)} = 128$ and $D^{(2)} = 32$ units, and Layer 3 having $D^{(3)} = 10$ units corresponding to the 10 possible digits, where $D^{(l)}$ is the number of units or dimensions of Layer l. In Section 1.6, we detail how to train this model. In practice, a CNN model is a better choice for digit classification; we use an MLP model to introduce this type of topology with a simple example.

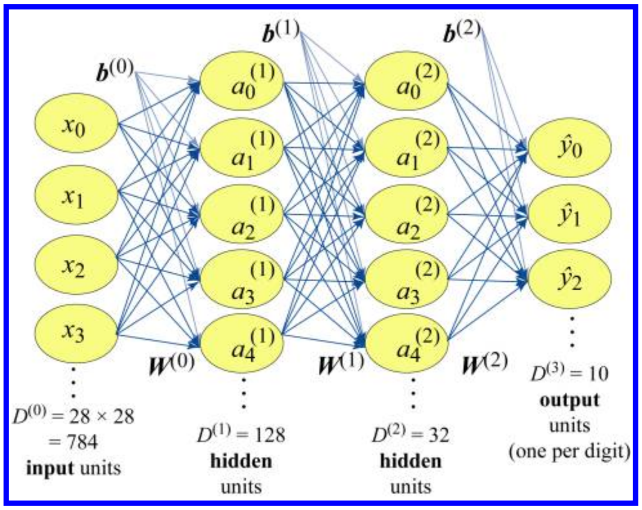


Figure 1.6: An MLP with four layers: the input layer, two hidden layers, and the output layer. This model maps the 784 pixel values to a probability distribution over 10 possible classes.

1.5.2 CONVOLUTIONAL NEURAL NETWORK

A CNN is a special type of FFNN widely used for computer vision applications, such as image classification, image detection, image similarity, semantic segmentation, human pose estimation, action recognition, and image feature extraction. Commercial applications include facial recognition, visual search, optical character recognition for document scanning, X-ray tumor detection, drug discovery, and MRI analysis. Figure 1.7 shows the input to a CNN and the output activations at each layer. Convolutional units are explained in detail in Section 2.3, and various CNN models used in production are discussed in Section 3.2.

CNNs are also used as image feature extractors; the output of one of the last layers (usually the second-to-last layer) is used as a feature vector representing the image. This vector becomes the input to other algorithms, such as an RNN to generate a textual description of the image, or to a reinforcement agent learning to play a video game, or to a recommender system that uses visual similarity.

1.5.3 RECURRENT NEURAL NETWORK

An RNN is a directed graph with nodes along a temporal or contextual sequence to capture the temporal dependencies. RNN models are used with sequential data common in language tasks and time-series forecasting. Commercial applications include stock price forecasting, text summarization, next-word recommendation, language translation, simple chatbot tasks, image description generation, speech recognition, and sentiment analysis.

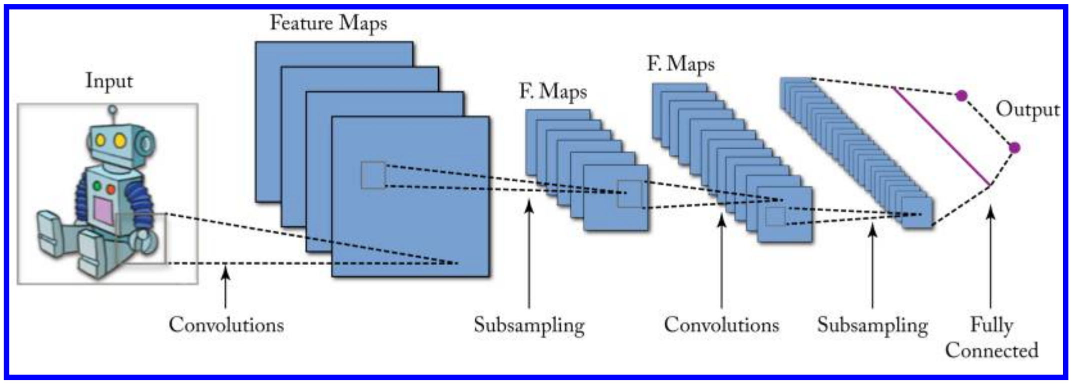


Figure 1.7: A CNN model with several layers maps the input image to a probability distribution across multiple possible labels. Source: [Wik15] (CC BY-SA 4.0).

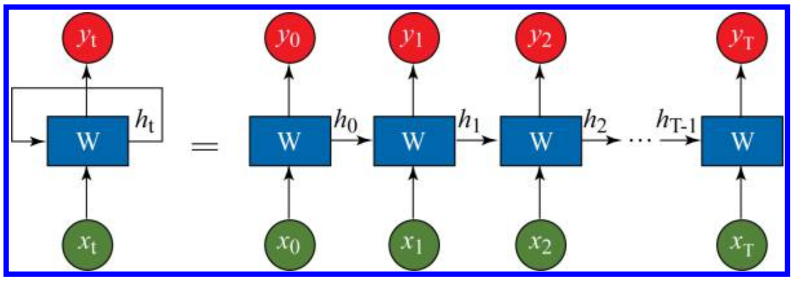


Figure 1.8: An RNN topology can be represented as an FFNN topology with the same weights **W** across all the layers.

The RNNs inputs and outputs can vary in length, unlike in MLP and CNN models. For instance, in machine translation, the input and output sentences have a different number of words. An RNN can be unrolled and represented as an FFNN sharing the same weights across the layers, as shown in Figure 1.8.

RNN models can be stacked with multiple layers and also bidirectional, as shown in Figure 1.9. The main building block of an RNN model is a recurrent unit that captures a representation or "memory" of the aggregated relevant data from previous steps. There are three main types of RNN models depending on the type of recurrent units they use: vanilla RNN, LSTM, and GRU units, detailed in Section 2.5. In the literature, the term *RNN* denotes either a *vanilla RNN* or, more broadly, these three types of models. In this book, when referring to a vanilla RNN model, we explicitly use the term vanilla RNN to prevent confusion. LSTM and GRU models are usually favored over vanilla RNN models for their superior statistical performance.

RNN models have two significant challenges: (1) capturing the dependencies in long sequences and (2) parallelizing the computation (due to the sequential nature where the output at a

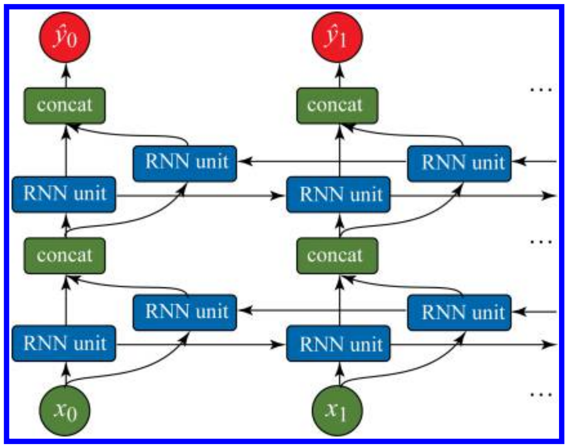


Figure 1.9: A bidirectional RNN model with two layers.

given timestep depends on the previous timestep). Using attention layers, detailed in Section 2.8, mitigates these challenges. Concatenating multiple sequential outputs from the first layer in the stack and passing those as inputs to the second layer in the stack improves the computational efficiency in a model with multiple layers [HSP+19].

1.5.4 TRANSFORMER NETWORKS

A transformer model learns how various parts of the input affects the output using an attention module. These models are also called attention-based models and have gained wide adoption for language tasks with similar applications to RNNs. They mitigate the challenges with RNNs discussed in the previous section at the expense of additional computations. The attention module can capture dependencies across long sequences. A transformer model consumes the entire sequence at once and uses multiple FFNNs in parallel together with attention modules to learn a set of weights corresponding to the influence between inputs and outputs [VSP+17]. For instance, in machine translation, the attention weights capture how each word in the output (target) language is influenced by both the neighboring words and the words in the input (source) language. The attention module is explained further in Section 2.8, and various transformer-based models used in production are discussed in Section 3.3.

1.5.5 GRAPH NEURAL NETWORK

NNs operate on data organized as vectors with MLPs, as grids or lattices with CNNs, and as sequences or chains with RNNs and Transformers. A GNN is a generalization of an MLP, CNN, RNN, and Transformer that operates on graphs rather than tensors, as shown in Figure 1.10. A graph is composed of nodes (also known as vertices) and edges representing the relation between the nodes. GNN nodes learn the properties of the neighboring nodes. Graphs are

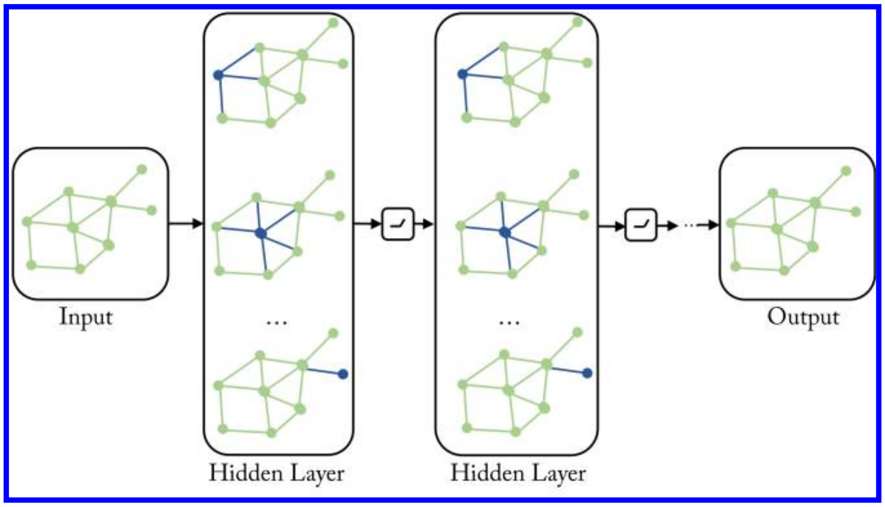


Figure 1.10: A GNN operates on graphs rather than tensors. This GNN has four layers, an input, output, and two hidden layers. Based on [Jad19].

common in many applications, such as in social networks to represent persons and their connections, in molecular biology to represent atoms and bonds, in recommender systems to represent users, items, and ratings, in telecommunications to represent networks, and in drug discovery to represent the compound structure and protein-enzyme interactions. Graphs of graphs are also common; one example is web document classification with a graph of web documents where the edges are the hyperlinks, and each node is a graph with XML-formatted elements for each document. GNNs provide the structure to learn and make predictions on graphs, often with sparsely labeled data. Given the sparse representation of the adjacency matrix in GNNs, it is beneficial to advance work in nonsequential memory access retrieval to accelerate GNNs.

GNNs were introduced in 2009 and have recently seen astronomical growth in academia [SGT+09]. Given the many real-world graph applications, rapid growth in the industry over the next few years is expected. Large-scale recommender systems, such as Pinterest's PinSage, already use GNNs [YKC+18]. Hyperscalers are developing platforms, such as Alibaba's AliGraph, Microsoft's NeuGraph, and Amazon's Deep Graph Library (DGL) to facilitate GNN industry adoption [ZZY+19, MYM+19, WVP+19]. PyTorch Geometric (PyG) is primarily targeting academic research [FL19].

1.5.6 ADVERSARIAL NETWORK

An AN or a generative adversarial network (GAN) consists of two subnetworks: a discriminator and a generator, as shown in Figure 1.11 [GPM+14]. During training, they compete in a minimax game. The generator learns to generate raw data with corresponding statistics to the training

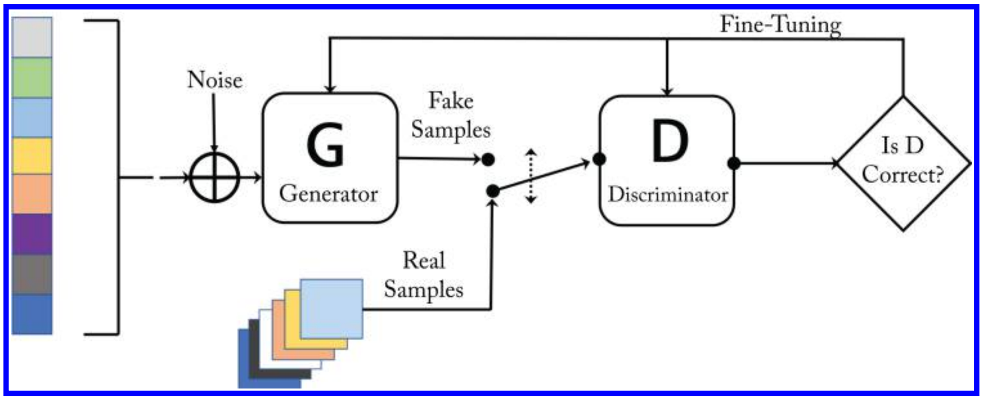


Figure 1.11: A generative adversarial network has a discriminator and a generator network that compete with each other. Based on [Gha17].

set. The discriminator evaluates the candidates as authentic or synthetic (generated). The generator's objective is to increase the error rate of the discriminator. It generates data to fool the discriminator into classifying it as authentic. The discriminator is initially trained with a training dataset. Then it is tuned as it competes with the generator. As the model trains, the generated data becomes more authentic-like, and the discriminator improves at recognizing synthetic data.

Yann LeCun, likely the most prominent DL scientist, described GANs as "the coolest idea in machine learning in the last twenty years" [Lec16]. GANs were initially proposed for unsupervised learning and now they are used across all types of learning. Wasserstein GAN (WGAN) improves the stability of learning the model, and Weng provides a detailed explanation of the mathematics used in WGAN [ACB17, Wen17].

GANs are also used for model physics-based simulations in particle physics and cosmology, reducing the simulation time by orders of magnitude [PdO+18, RKL+18]. Section 3.2.5 discusses various GANs use for image generation.

1.5.7 AUTOENCODER

An AE is a class of unsupervised learning topology that learns a low-dimensional representation (an encoding) of the input. The AE learns to reconstruct the input data in the output layer and uses the output of the bottleneck layer (usually the middle-most layer) as the low-dimensional representation. The number of units typically decreases in each layer until the bottleneck layer, as shown in Figure 1.12.

AEs are used (1) as a compact (compressed) representation of the input data; (2) as a preprocessing step to a classification problem where the data is first encoded and then passed to the classifier; (3) in a data matching problem by comparing the encoding of two data samples; (4) to

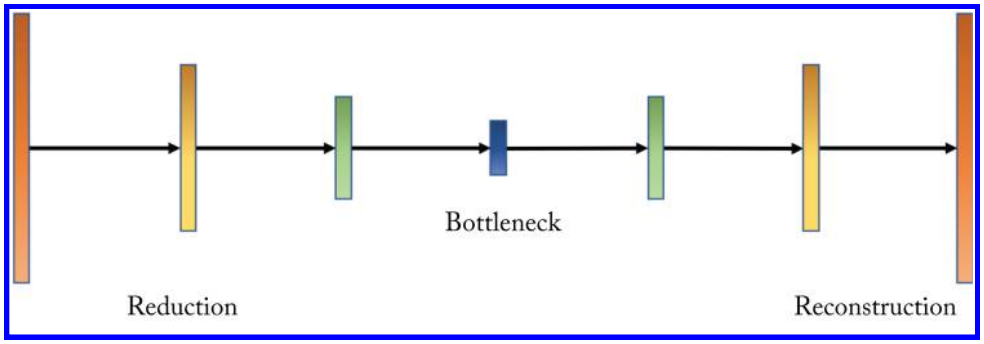


Figure 1.12: An autoencoder learns to reconstruct the input data in the output layer. The output of the bottleneck layer is often used as a low-dimensional representation of the input.

denoise data by learning a mapping from a noisy input to a clean output; and (5) as a generative model to generate data using the decoder (known as a variational autoencoder (VAE)).

1.5.8 BAYESIAN NEURAL NETWORKS

A BNN combines the strength of a NN and a Bayesian model to estimate the uncertainty of a NN prediction [Nea95]. Typical NNs use single value weights, whereas BNNs use a probability distribution over each weight; that is, BNNs provide an estimate of each weight's uncertainty, which can be used for performance guarantees and to improve interpretability. However, analytically computing and updating each distribution is prodigiously expensive. Approximating the prior and posterior distribution is an active area in research with variational inference as a common algorithm (discussed elsewhere) [BCK+15]. Despite their popularity in academia, due to their current lack of adoption in production, BNNs are not covered further.

1.5.9 SPIKING NEURAL NETWORKS

An SNN is inspired by the way natural neurons transmit information using spikes [NMZ19]. SNNs represent a whole different class of NNs differentiated by their local learning rules and are often not included in DL literature. The primary advantage of SNNs is the potential for lower power consumption using a specialized hardware known as a neuromorphic processor, such as Intel's Loihi, IBM's TrueNorth, and aiCTX's Dynamic Neuromorphic Asynchronous Processor (DYNAP) processors [RJP19]. SNNs are currently not used in production due to their inferior statistical performance and limited applications compared to other types of NNs, and therefore are not discussed further.

1.6 TRAINING AND SERVING A SIMPLE NEURAL NETWORK

A NN topology consists of the number of layers, units in each layer, and activation functions per layer. Training a model requires selecting and tuning the following hyperparameters: the NN topology, the methodology to initialize the weights, the objective function, the batch size, and the optimization algorithm and corresponding learning rate (LR). Note that in the DL literature (and in this book), *hyperparameters* are the knobs tuned by the data scientist, and *parameters* are the model weights. The type of topologies used across various workloads are discussed in Chapter 3, and the training steps are introduced below and detailed in Chapter 4. Preparing the training dataset and training with imbalanced datasets where the training samples are not evenly distributed among the classes are discussed in Section 4.5, and methods that may help identify some biases in training datasets are discussed in Sections 10.4 and 10.5. Software libraries like TensorFlow and PyTorch facilitate the training and serving of NNs and are discussed in Chapter 9. Distributed training across multiple nodes can reduce the total time-to-train (TTT), is detailed in Chapter 5.

A training system aims to reduce the time to train without sacrificing accuracy. A serving or inference system aims to maximize the throughput constrained by a latency requirement. Product recommendations, search results, voice assistant queries, and pedestrian identification in autonomous vehicles, require real-time (low latency) results. Typically, only one data sample or a micro-batch is used at a time to meet the particular application's latency requirement. Given the fewer computations per byte read from memory, the operational intensity or compute efficiency in GPUs and CPUs is lower in serving than in training.

A nomenclature note: in this book, a batch (sometimes called *mini-batch* in the literature) refers to a subset of training samples ranging from 1 to the entire dataset. A *full-batch* refers to a batch composed of the entire training dataset. A *micro-batch* refers to a batch with 1–8 samples. A *large-batch* refers to a batch size greater than 1,000 samples but less than the entire training dataset. A *node-batch* refers to the batch processed in a single node during distributed training, discussed in Chapter 5.

In the remainder of this section, we introduce some components of NNs and describe the training process using a simple example. The primary compute operations in training and serving a model are multiplications and additions, which are typically computed in groups and represented as matrices.

Once a topology is defined, training involves learning a good set of weight values. The training steps for supervised learning are typically as follows:

1. Initialize the weights or parameters of the model typically by sampling from a zero-mean Gaussian or uniform distribution.

2. Forward propagate a training sample or, more commonly, a batch of samples through the network to compute the output.

3. Evaluate the cost or penalty using a metric of difference between the expected outputs (known from the training labels) and the actual outputs.

4. Backpropagate the gradient of the cost with respect to each layer's weights and activations.

5. Update the weights of the model using the computed gradients.

6. Return to Step 2, or stop if the validation error is less than some threshold or is not decreasing.

During training, the dataset is processed in batches. The completion of a cycle through steps 2–6 for a batch is called an *iteration*, and each cycle through the entire training dataset is called an *epoch*. For instance, if the dataset has $1M$ samples and a batch has 100 samples, it takes $10K$ iterations to complete an epoch.

Training a model may require tens of epochs to learn a good set of weights. After training, the validation (also called out-of-sample) performance is measured using a validation dataset. The validation dataset contains labeled data not used during training and should be as similar as possible to the serving data the model encounters when deployed. The performance on this validation dataset is a good indicator of the performance in deployment and helps to determine if the model overfits the training dataset. Overfitting occurs when a model learns features unique to the training data and, therefore, does not generalize to data outside the training dataset. Regularization techniques to mitigate overfitting are discussed in Section 4.1.

During serving, the model processes a micro-batch. The data is propagated forward through the network to compute the output. Serving is also known as inference since the model is inferring the label of the data sample. Step 2 above is inference; that is, inference is a step in the training process but usually with a smaller batch size and some optimizations specific to serving.

The following example illustrates the training process. The task is to classify handwritten digits from the MNIST dataset using an MLP model [LBB+98]. Figure 1.13 shows a small subset of the 70,000 gray-scaled 28×28 pixel images in the MNIST dataset. Typically with MNIST, 60,000 images are used for training and 10,000 images are used for validation. In practice, a CNN model would be a better choice for image classification, but a simple MLP model is used to introduce some fundamental concepts.

Each layer in the MLP is composed of units (neurons) that linearly combine the weighted outputs or activations from the previous layer plus a bias weight, as shown in Figure 1.14 for one unit. The output from this affine transformation is passed to a nonlinear activation function $g(\cdot)$. An *activation function* refers to the nonlinear function, an *activation input* is the input to the activation function, and an *activation* (short for *activation output*) refers to the output of an activation function. Common activation functions are the rectified linear unit (ReLU) and variants of ReLU, the sigmoid and its generalization, the softmax, and the hyperbolic tangent (tanh), which are all detailed in Section 2.1.

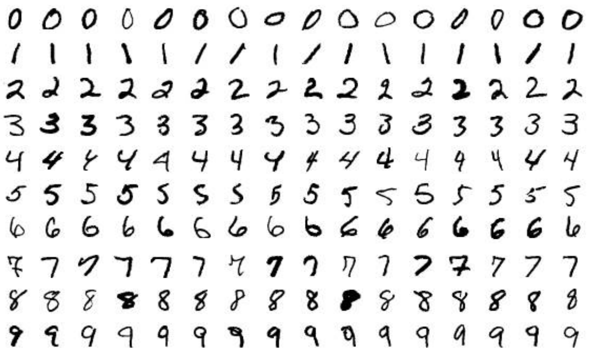


Figure 1.13: Examples from the MNIST dataset. Each digit image has 28×28 pixels. Source: [Wik17] (CC BY-SA 4.0).

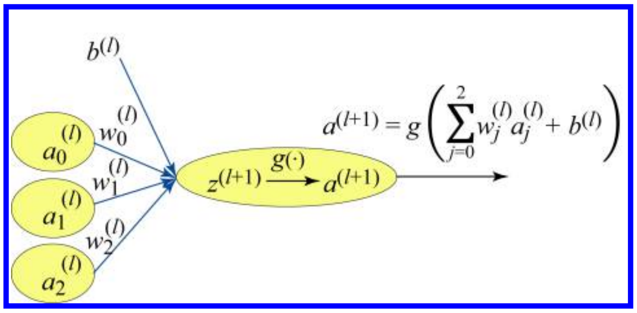

$b^{(l)}$
$w_0^{(l)}$
$w_1^{(l)}$
$w_2^{(l)}$
$a_0^{(l)}$
$a_1^{(l)}$
$a_2^{(l)}$
$z^{(l+1)} \xrightarrow{g(\cdot)} a^{(l+1)}$
$a^{(l+1)} = g\left(\sum_{j=0}^{2} w_j^{(l)} a_j^{(l)} + b^{(l)}\right)$

Figure 1.14: A neural unit at layer $(l+1)$ applies a nonlinear transformation or function to the weighted sum of the activations from the previous layer (l).

The MLP model used for this digit classification task, shown in Figure 1.6, has 784 units in the input layer (Layer 0) corresponding to the number of pixel values in each image. The output layer has 10 units corresponding to the probability distribution of the possible 0–9 labels. This MLP has two hidden layers with 128 and 32 units, respectively. The choice for the number of hidden layers and the number of units in each layer requires experimentation. In Section 4.5, we discuss techniques to choose an appropriate topology.

To train the model, the 28×28 image pixel values are reordered as a 784×1 vector and normalized to zero-mean and unit-norm (the benefits of normalization are explained in Section 2.6). This is the input to the NN and can be thought of as the activations of Layer 0. The input $z_i^{(1)}$ to unit i in Layer 1 is the weighted sum of the activations of Layer 0 plus a bias. The activation $a_i^{(1)}$ of unit i is a nonlinear transformation of the unit's activation input $z_i^{(1)}$:

$$a_i^{(1)} = g\left(z_i^{(1)}\right) = \max\left(0, z_i^{(1)}\right),$$

where $g(\cdot)$ is the ReLU activation function, and

$$z_i^{(1)} = \sum_{k=0}^{783} w_{ik}^{(0)} x_k + b_i^{(0)}$$

is the output of the affine transformation (also known as the activation input in Layer 1), where x_k represents the $k \in [0, 783]$th pixel value. In this example, the activation functions are ReLU for Layers 1 and 2, and softmax for the output layer. The ReLU function zeros out negative values and keeps the positive values unchanged. The softmax function is used in the output layer to map a vector of values to a probability distribution where the values are all between 0 and 1 and sum to 1. The ith output value can be computed as follows:

$$\hat{y}_i = \frac{\exp\left(z_i^{(3)}\right)}{\sum_{k=0}^{9} \exp\left(z_k^{(3)}\right)},$$

where $\hat{y}_i$ represents the probability the input image corresponds to class i. There is no bias term in a softmax layer.

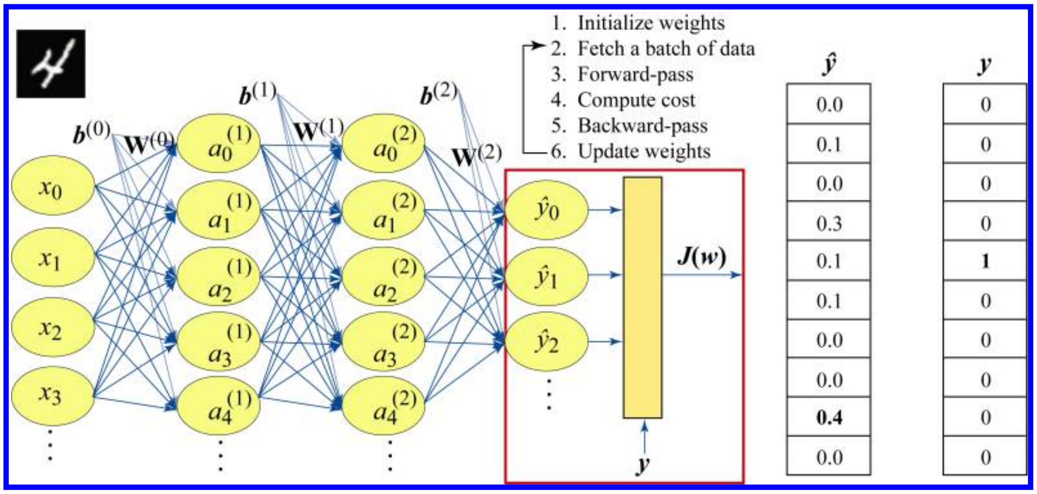
This softmax output is compared with the ground truth. For this task, the ground truth is a one-hot vector with the nonzero index corresponding to the correct class label. The cross-entropy loss is:

$$-\sum_{k=0}^{9} y_k \log(\hat{y}_k),$$

where log represents the natural logarithm (log base-e), y_k is 1 if the sample belongs to class $k \in [0, 9]$ and 0 otherwise, and $\hat{y}_k$ is the model's prediction (as a probability) that the sample belongs to class k. Figure 1.15 depicts the expected and actual output for a sample image corresponding to the digit 4. In the figure, the model's output $\hat{\mathbf{y}}$ incorrectly indicates digit 8 is the most likely inferred interpretation. Additional training iterations are needed to reduce this loss.

The gradients of the cost with respect to all the layers' activations and weights are computed using the chain rule from the last layer and moving backward layer by layer toward the first layer. Hence, the name backpropagation. The gradients provide a measurement of the contribution of each weight and activation to the cost. In practice, all of the activations for a given batch and a given layer are simultaneously computed using matrix algebra. For these computations, data scientists use software libraries optimized for the particular hardware target.

During training, the activations are saved for the backpropagation computations. Therefore, hardware for training requires a larger memory capacity than hardware for inference. The required memory is proportional to the batch size.



1. Initialize weights
2. Fetch a batch of data
3. Forward-pass
4. Compute cost
5. Backward-pass
6. Update weights
$\hat{y}$
y
0.0 0.1 0.0 0.3 0.1 0.1 0.0 0.0 0.4 0.0
0 0 0 0 1 0 0 0 0 0
$b^{(0)}$ $W^{(0)}$ $b^{(1)}$ $W^{(1)}$ $b^{(2)}$ $W^{(2)}$
x_0 x_1 x_2 x_3
$a_0^{(1)}$ $a_1^{(1)}$ $a_2^{(1)}$ $a_3^{(1)}$ $a_4^{(1)}$
$a_0^{(2)}$ $a_1^{(2)}$ $a_2^{(2)}$ $a_3^{(2)}$ $a_4^{(2)}$
$\hat{y}_0$ $\hat{y}_1$ $\hat{y}_2$
$J(w)$
y

Figure 1.15: A batch of size 1 containing a sample image of the digit 4 is passed through the model. The actual output $\hat{\mathbf{y}}$ and the expected output (ground truth) $\mathbf{y}$ are used to compute the cost $J(\mathbf{w})$. The model performs poorly in this example and predicts digit 8 with 40% probability and digit 4 with 10% probability. The cross-entropy loss is $-\log(0.1)$.

1.7 MEMORY AND COMPUTATIONAL ANALYSIS

The training process requires memory for (1) the model weights, (2) all the activations (including the batch of input data), and (3) two consecutive gradient activation layers used for gradient computations. The serving process requires memory for (1) the model and (2) two consecutive activation layers (including the input batch).

The number of weights N_w, including the biases, in the MLP model in the previous section is:

$$\begin{aligned} N_w &= N_{w_{L_0}} + N_{w_{L_1}} + N_{w_{L_2}} \\ &= (784 \times 128 + 128) + (128 \times 32 + 32) + (32 \times 10 + 10) \\ &= 104{,}934. \end{aligned}$$

This small model requires 420 KB of memory if 4 bytes are used to represent each weight. Note that in some literature, a based-2 metric is used, where a KiliByte (KiB), MiliByte (MiB), and GibiByte (GiB) represents 2^{10}, 2^{20}, and 2^{30} bytes, respectively. Thus, 420 KB is approximately 410 KiB.

The total number of activations N_a is the sum of the activations in each layer:

$$\begin{aligned} N_a &= N_{a_{L_0}} + N_{a_{L_1}} + N_{a_{L_2}} + N_{a_{L_3}} \\ &= (784 + 128 + 32 + 10) \times N \\ &= 954N, \end{aligned}$$

where N is the number of images in each batch. The size of the two largest gradient activation layers N_g required for the gradient computations, is:

$$\begin{aligned} N_g &= N_{a_{L_1}} + N_{a_{L_2}} \\ &= (128 + 32) \times N \\ &= 160N. \end{aligned}$$

Thus, the total memory requirement for training, using 4 bytes for each value, is:

$$\begin{aligned} T_M &= (N_w + N_a + N_g) \times 4 \\ &= (104{,}934 + 1114N) \times 4 \\ &= 419736 + 4456N. \end{aligned}$$

Assuming a batch of $N = 128$, the required memory for training is 1.0 MB.

The total memory requirement for inference, using 4 bytes for each value, is:

$$\begin{aligned} T_M &= (N_w + N_a) \times 4 \\ &= (104{,}934 + (784 + 128)N) \times 4 \\ &= 419736 + 3648N. \end{aligned}$$

Assuming a batch of $N = 1$, the require memory for inference is 424 KB.

1.8 HARDWARE DESIGN CONSIDERATIONS

The primary components in a DL platform are multitudinous multiplication and addition units, sufficient memory capacity, high memory bandwidth to feed the compute units, high inter-node bandwidth for distributed computing, and power to operate. Processing state-of-the-art models is increasingly mandating more of these components. Designing hardware requires carefully balancing these components across a huge space of numerical formats, storage and memory hierarchies, power limitations, area limitations, accuracy requirements, hardware- or software-managed caches or scratchpads, support for dense and sparse computations, domain-specific to general-purpose compute ratio, compute-to-bandwidth ratios, and inter-chip interconnects. The hardware needs the flexibility and programmability to support a spectrum of DL workloads and achieve high workload performance. In this section, we introduce some of these components and expand upon them in Chapter 7.

The core compute of training and serving are multiplications and additions. Compute is inexpensive relative to main memory bandwidth and local memory. Moore's Law continues to deliver exponential growth in the number of transistors that can be packed into a given area. Thus, the silicon area required for a set of multiply-accumulate (MAC) units is decreasing. While hardware companies often highlight the theoretical maximum number of operations (ops) per second (ops/s or OPS), the most significant bottlenecks are typically the main memory bandwidth and the local memory capacity. Without sufficient bandwidth, the overall compute

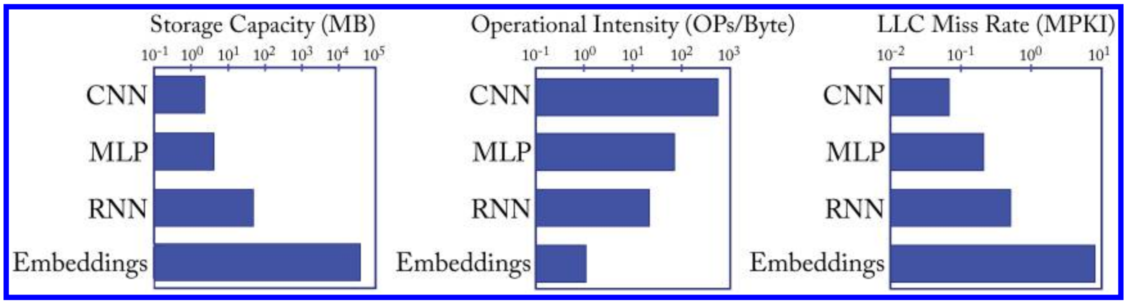


Figure 1.16: Typical CNNs, MLPs, RNNs, and embeddings differ by orders of magnitude in storage, operational intensity, and memory access irregularities. Based on [Haz20].

efficiency or utilization (the percentage of used compute cycles vs. the total compute capacity) is low for workloads bottlenecked by bandwidth (also known as bandwidth bound), and adding more compute capacity does not improve their performance. Keeping the data close to the compute can alleviate this bottleneck. In order of decreasing access time and increasing die area, the storage types are nonvolatile memory (flash memory, magnetic disk), DRAM (HBM2/E, GDDR6, DDR4, LPDDR4/5), SRAM (scratchpad, cache), and registers. DRAM is often called main memory and SRAM local memory.

The design of a balanced platform is complicated by the spectrum of workloads with diverse compute, memory, and bandwidth requirements. For instance, the CNNs, MLPs, RNNs, and embeddings used at Facebook (and similar at other hyperscalers) differ by orders of magnitude in these requirements, as shown in Figure 1.16 [Haz20]. Operational intensity is a measure of the number of operations performed per byte read from memory. The last level cache (LLC) miss rate as measured by misses per 1000-instructions (MPKI) is a standard metric to analyze the local memory (SRAM)'s efficient use and can be a metric for the irregular memory access patterns of a workload.

The numerical format is another design consideration that can impact the computational (speed) performance and statistical (accuracy) performance. Figure 1.17 shows various numerical formats, detailed in Section 6.1. A numerical representation with fewer bytes can improve the number of operations per cycle and reduce power consumption but may result in lower statistical performance. Training uses single-precision floating-point (*fp*32) with half-precision floating-point (*fp*16) and bfloat16 (*bf*16) rapidly gaining adoption. Inference uses *fp*16 and *bf*16 with 8-bit integer (*int8*) gaining adoption for some applications. A research area is developing numerical representations that can better represent values using 8 bits, such as *fp*8, discussed in Section 6.1, and can be efficiently implemented in silicon. Other techniques to reduce the memory and bandwidth requirements are increasing the sparsity and compressing the data.

A MAC unit computes the product of two values and aggregates the result to a running sum of products. The numerical format of the output (the accumulation) may be different

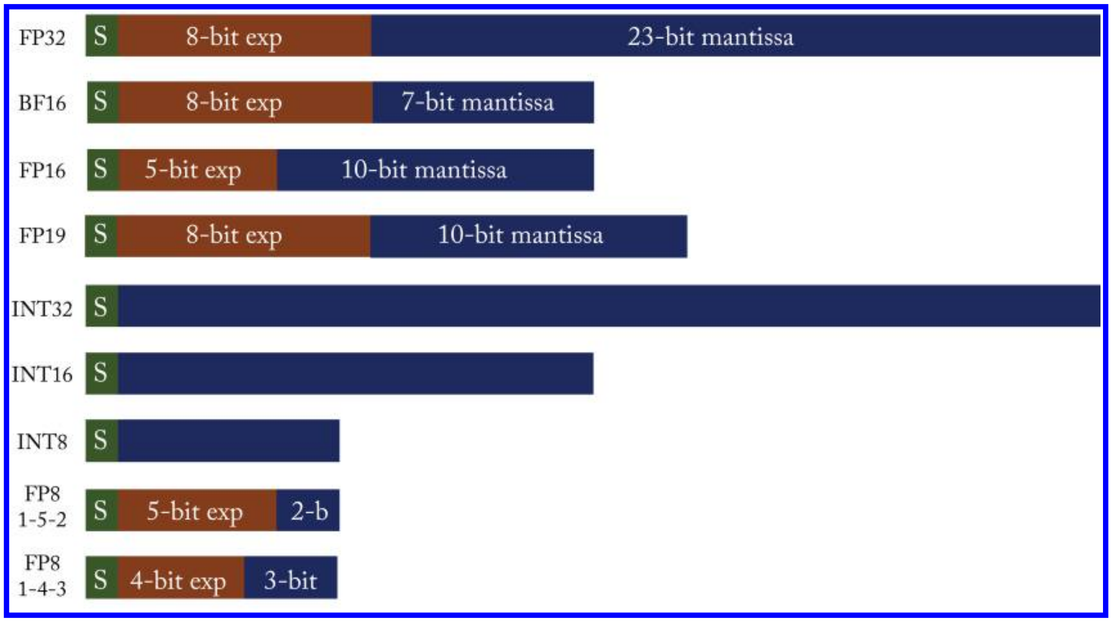


Figure 1.17: Numerical formats. Green is the sign bit. Brown are the exponent bits. Blue are the mantissa bits.

from the input. Computations involving dot products, such as in matrix multiplications and convolutions, typically use MACs. When describing MAC units, the notation used is *MAC-input-format* $\rightarrow$ *MAC-accumulate-format*. For instance, *int8* $\rightarrow$ *int32* means the *int8* values are multiplied and accumulated as *int32* values. Accumulating values in a large numerical format mitigates numerical overflows.

Different hardware usages have different requirements. Table 1.1 shows the high-level requirements for common usages by hyperscalers: topology design, training established production models (Trn. Prod.), data center inference (Inf. DC), and edge inference (Inf. Edge). In the table, format refers to the number of bits to represent the weights and activations. Training requires more memory and bandwidth than inference to transfer and store the activations. Another use case not shown in Table 1.1 is for hardware design, which requires reconfigurable hardware (for example, FPGAs) or hardware simulators.

1.9 SOFTWARE STACK

Software is critical to a DL system. The software stack is organized as follows:

- deployment and training management systems;
- frameworks, inference engines, and graph compilers;

Table 1.1: Hardware characteristics according to usage

Usage	Format (bits)	Comp	Main Mem	Mem BW	Programmability	Internode	Interserver
Design	32	High	High	High	High	Yes	Some
Trn. Prod.	16 & 32	High	High	High	Mid	Yes	Yes
Inf. DC	16 & 8	Mid	Mid	Mid	Mid	Some	No
Inf. Edge	8	Low	Low	Low	Low	Some	No

- DNN primitive libraries and tensor compilers;
- instruction set architecture (ISA); and
- operating systems.

The primary software stack design goals are ease-of-use and high performance across various models and hardware devices.

A deployment and training management system facilitates taking a model across the pipeline stages: data preparation, topology exploration, experiment tracking, model packaging, at-scale model deployment, and retraining. The management system is designed to meet the needs of the data scientist and the infrastructure team. It provides a collaborative and secure environment, and access to the latest ML libraries, such as TensorFlow and PyTorch.

At the core of the software stack are compilers to transform the programmer's high-level code into executable code that runs efficiently on a target device. Frameworks and inference engines (IEs), such as TensorFlow, PyTorch, OpenVINO, and TensorRT, provide a high-level abstraction to the operators used across DL models. They use graph optimizers (either built-in or external) to optimize the model. The framework's scheduler relies on low-level DL and Math libraries, such as oneDNN (formerly called Intel MKL-DNN), Nvidia cuDNN, Eigen, or OpenBLAS, or in tensor compilers for optimizations to standard DL functions. Frameworks also have a code generation path to supplement these libraries with other compilers, such as LLVM.

The ISA defines the operators, data types, and memory management for an abstract computer architecture. A particular implementation of an ISA is called a *microarchitecture*. For instance, Intel and AMD CPUs use the x86 or x86-64 ISA across different microarchitecture implementations and CPU generations. Programs are binary compatible across all microarchitecture implementations of a particular ISA. Different microarchitectures can have different properties that can affect their performance, such as instructions latencies and cache hierarchies. A specific microarchitecture can be available in various flavors with different frequencies and cache sizes.

The operating system (OS) manages all the hardware and software in a compute device; it allocates hardware resources, such as compute and memory, to the software applications. An overview of operating systems is beyond the scope of this book.

Chapter 8 introduces programming languages and compiler techniques, and Chapter 9 details the prevalent DL graph and tensor compilers. Chapter 10 highlights higher-level platforms used by hyperscalers to manage training and deployment.

1.10 NOTATION

This section references the notation used throughout this book to represent input data, labels, weights, affine transformations, activations, and outputs. Recall that the compute operations in training and serving boil down to multiplications and additions. Linear algebra is used to represent groups of multiplications and additions as a single matrix-matrix or matrix-vector or vector-vector operation. While helpful, a background in linear algebra is not required; the reader can overlook the equations without a significant impact on the other parts of the book.

In DL literature, the output from an affine transformation can be equivalently represented as either

$$z_j^{(l+1)} = \sum_{i=0}^{D^{(l)}-1} w_{ji}^{(l)} a_i^{(l)} + b_j^{(l)}$$

or as

$$z_j^{(l+1)} = \sum_{i=0}^{D^{(l)}} w_{ji}^{(l)} a_i^{(l)},$$

where the bias term $b_i^{(l)}$ is included in the second equation as an additional weight with a corresponding $a_{D(l)}^{(l)} = 1$ appended to the activations. In this book, we use the first notation and explicitly represent the bias separately. The addition notation used is as follows:

- Superscripts in parenthesis means layer number
- Superscripts in brackets means sample number
- Subscript represents indices in matrices or vectors
- Bold-font lowercase represents vectors
- Bold-font uppercase represents matrices
- $\mathbf{x}^{[n]}$ and $\mathbf{y}^{[n]}$: input features and expected output (ground-truth), respectively, for the nth sample
- $(\mathbf{x}^{[0]}, \mathbf{y}^{[0]}), \ldots, (\mathbf{x}^{[N-1]}, \mathbf{y}^{[N-1]})$: training data with N samples
- $y \in \{0, 1\}$: for binary classification

- $\mathbf{y} \in \Re^M$: typically a vector with a one at the entry corresponding to its class assignment and zeros everywhere else for M-nary ($M > 2$) classification
- $\hat{\mathbf{y}} = f_{\mathrm{w}}(\mathbf{x}) \in \Re^M$: output of the model
- $D^{(l)}$: number (dimensions) of units at Layer $l \in [0, L-1]$, where L is the number of layers (note that $D^{(L-1)} = M$ for M-nary classification)
- $\mathbf{W}^{(l)} \in \Re^{D^{(l+1)} \times D^{(l)}}$: weights (not including the biases) from Layer l to Layer $l+1$, where

$$\mathbf{W}^{(l)} = \begin{bmatrix} w_{00}^{(l)} & w_{01}^{(l)} & w_{02}^{(l)} & \cdots & w_{0(D^{(l)}-1)}^{(l)} \\ w_{10}^{(l)} & w_{11}^{(l)} & w_{12}^{(l)} & \cdots & w_{1(D^{(l)}-1)}^{(l)} \\ \vdots & \vdots & \vdots & & \vdots \\ w_{(D^{(l+1)}-1)0}^{(l)} & w_{(D^{(l+1)}-1)1}^{(l)} & w_{(D^{(l+1)}-1)2}^{(l)} & \cdots & w_{(D^{(l+1)}-1)(D^{(l)}-1)}^{(l)} \end{bmatrix}$$

- $w_{ji}^{(l)} \in \mathbf{W}^{(l)}$: weight from output i in Layer l to input j in Layer $l+1$, where $i \in [0, D^{(l)} - 1]$, and $j \in [D^{(l+1)} - 1]$
- $\mathbf{a}^{(l)} = g(\mathbf{z}^{(l)}) \in \Re^{D^{(l)}}$: activation of Layer $l \in [0, L-1]$
- $\mathbf{a}^{(0)} = \mathbf{x}$: NN input (usually normalized)
- $\mathbf{z}^{(l)} \in \Re^{D^{(l)}}$: activation inputs to Layer $l \in [1, L-1]$
- $\mathbf{z}^{(l+1)} = \mathbf{W}^{(l)}\mathbf{a}^{(l)} + \mathbf{b}^{(l)} = [\mathbf{W}^{(l)}\ \mathbf{b}^{(l)}] \times [\mathbf{a}^{(l)}; 1]$, where $[\mathbf{W}^{(l)}\ \mathbf{b}^{(l)}]$ represents a matrix with $\mathbf{b}^{(l)}$ right appended to matrix $\mathbf{W}^{(l)}$, and $[\mathbf{a}^{(l)}; 1]$ represents a vector with a 1 bottom appended to vector $\mathbf{a}^{(l)}$
- $\mathbf{X} = [\mathbf{x}^{[0]}, \cdots, \mathbf{x}^{[N-1]}] \in \Re^{D^{(0)} \times N}$
- $\mathbf{Y} = [\mathbf{y}^{[0]}, \cdots, \mathbf{y}^{[N-1]}] \in \Re^{M \times N}$
- $\hat{\mathbf{Y}} = [\hat{\mathbf{y}}^{[0]}, \cdots, \hat{\mathbf{y}}^{[N-1]}] \in \Re^{M \times N}$
- $\mathbf{Z}^{(l)} = [\mathbf{z}^{(l)[0]}, \cdots, \mathbf{z}^{(l)[N-1]}] \in \Re^{D^{(l)} \times N}$
- $\mathbf{A}^{(l)} = [\mathbf{a}^{(l)[0]}, \cdots, \mathbf{a}^{(l)[N-1]}] \in \Re^{D^{(l)} \times N}$
- $\mathbf{Z}^{(l+1)} = \mathbf{W}^{(l)}\mathbf{A}^{(l)} + [\mathbf{b}^{(l)} \cdots \mathbf{b}^{(l)}] = [\mathbf{W}^{(l)}\ \mathbf{b}^{(l)}] \times [\mathbf{A}^{(l)}; \mathbf{1}]$

CHAPTER 2
Building Blocks

There are four main types of NN topologies used in commercial applications: multilayer perceptrons (MLPs), convolution neural networks (CNNs), recurrent neural networks (RNNs), and transformer-based topologies. These topologies are *directed graphs* with nodes and edges, where a node represents an operator, and an edge represents a data-dependency between the nodes, as shown in Figure 1.5.

A node, also called primitive (short for primitive function), layer, expression, or kernel, is the building block of a topology. While the number of functions developed by researchers continues to grow, for example, the popular TensorFlow framework supports over 1,000 operators, the number of functions used in commercial applications is comparatively small. Examples of these functions are ReLU, sigmoid, hyperbolic tangent, softmax, GEMM, convolution, and batch normalization.

There are three types of compute functions: dense linear functions (e.g., GEMM and convolution), nonlinear functions (e.g., ReLU and sigmoid), and reduction functions (e.g., pooling). A dense linear function is typically implemented as a matrix-wise operator and a nonlinear function as an element-wise operator. A reduction function reduces the input vector to one scalar value.

Matrix-wise operators are compute-intensive and (depending on the hardware and the amount of data reuse) can be compute bound (referred to as Math bound in some GPU literature). Element-wise operators are compute-light and memory bandwidth bound. The inputs to these functions are read from memory, and the results are written back to memory; there is no data reuse.

A common technique to improve the compute efficiency of a model is to fuse a compute-light element-wise operator into a compute-intensive matrix-wise operator. Thus, the intermediate results are not written to and then read from main memory. The element-wise computations happen immediately after the matrix-wise computations while the data is in the registers or the storage closes to the computing unit. Chapter 8 details this and other techniques to improve the efficiency via software optimizations.

In this and the next chapter, we follow a bottom-up approach. In this chapter, we introduce the standard primitives in popular models used at hyperscalers. In the next chapter, we discuss the actual models and applications built using these primitives. Readers that prefer a top-down approach may first read Chapter 3 to better understand the types of models and applications

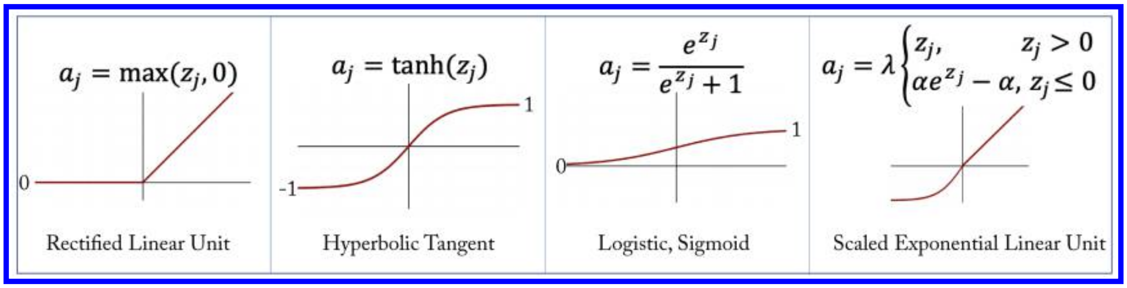


Figure 2.1: Examples of some activation functions $a_j = g(z_j)$ used in DL workloads.

before diving into the building blocks in this chapter. A review of the notation introduced in Section 1.10 can help understand the equations presented in this chapter.

2.1 ACTIVATION FUNCTIONS

An activation function is a nonlinear function applied to every element of a layer's input tensor. The most popular activation function is the rectified linear unit (ReLU). The ReLU function and its gradient are inexpensive to compute. Some models converge faster when the ReLU function is used [KSH12]. ReLU also increases sparsity which may provide computational and memory savings [GBB11].

The main drawback of ReLU is that the gradients are zero for negative activations, and the corresponding weights do not change during backpropagation. This is known as dying ReLU, and has motivated variants of ReLU, such as the Leaky ReLU (LReLU), Parametric ReLU (PReLU), Scaled Exponential Linear Unit (SELU), and the Gaussian Error Linear Unit (GELU) adopted in some attention-based models [HZR+15, KUM+17, HG16]. Another variant is ReLU6, which limits the maximum ReLU output to 6 and may improve the statistical performance when using a small numerical representation. These variants do not always result in superior statistical performance, and experimentation is required to assess the benefits.

The k-Winners-Take-All (k-WTA) activation function keeps the largest k activations in a layer and zeros out the reminder ones. A different k is typically chosen for each layer to maintain a level of constant sparsity ratio (e.g., 80%) across the layers [XZZ20].

The sigmoid and hyperbolic tangent (tanh) activation functions, shown in Figure 2.1, are commonly used in RNN models. Their main drawback is that for large positive or negative activations, the gradient is very small. During backpropagation, the product of various small gradients results in vanishing values due to the limited numerical representations in computers. This is known as the vanishing gradient problem. Variants of RNN models less prone to vanishing gradients are discussed in Section 2.5.

A benefit of the hyperbolic tangent over the sigmoid function is that it maintains a zero-mean distribution. It should be used over sigmoid except when the desired output is a probability distribution. The sigmoid function is commonly used as the activation function for the output layer in binary classification as the output is between 0 and 1 and can represent a probability.

The softmax is a generalization of the sigmoid and used for multiclass classification. It uses both element-wise and reduction operators. The output is interpreted as probability distribution with all the values between 0 and 1 and summing to 1. The ith output value can be computed as:

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{k=0}^{M-1} e^{z_k}},$$

where M is the number of classes. The activation input $\mathbf{z}$ to the softmax layer is called the *logit* vector or score, which corresponds to the unnormalized model predictions, and should not be confused with the logit (sigmoid) function.

Applying the exponential function to large logits magnifies the numerical errors. Therefore, it is a common practice to subtract the maximum logit m from all the logits before using the softmax function [BHH20]. The result is mathematically equivalent:

$$\frac{e^{x-m}}{e^{x-m}+e^{y-m}+e^{m-m}} = \frac{e^{x}e^{-m}}{(e^{x}+e^{y}+e^{m})e^{-m}} = \frac{e^{x}}{e^{x}+e^{y}+e^{m}},$$

where x, y, and m are three logits.

2.2 AFFINE

An affine transformation (also known as fully-connected, feedforward, or GEMM layer) provides a weighted sum of the inputs plus a bias. Figure 2.2 illustrates an affine transformation

$$z_j^{(l+1)} = \sum_{i=0}^{D^{(l)}-1} w_{ji}^{(l)} a_i^{(l)} + b_j^{(l)},$$

and the subsequent nonlinear activation

$$a_j^{(l+1)} = g\left(z_j^{(l+1)}\right).$$

An affine transformation can be formulated as a general matrix multiply (GEMM) for all the samples in a batch and for all the units in a layer, as shown in the last equation in Section 1.10. An affine transformation is called a linear primitive in DL literature (slightly abusing the term since a linear function should not have a bias).

Using a bias is always recommended even in large networks where a bias term may have a negligible impact on performance; removing the bias has little computational or memory savings. Note that when the affine layer is followed by a batch normalization (BN) layer (discussed in Section 2.6), the bias has no statistical impact as BN cancels out the bias.

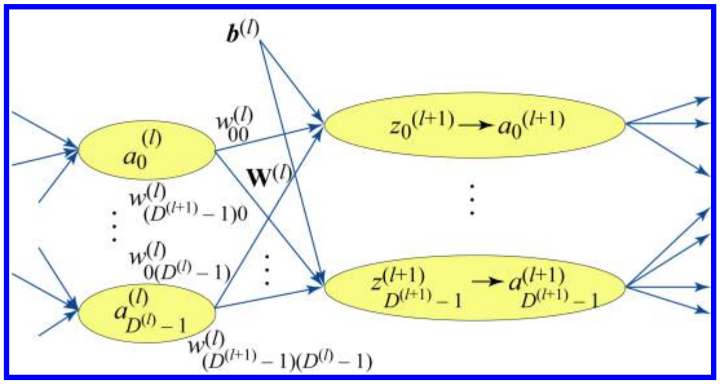

$b^{(l)}$
$a_0^{(l)}$
$w_{00}^{(l)}$
$W^{(l)}$
$w_{(D^{(l+1)}-1)0}^{(l)}$
$w_{0(D^{(l)}-1)}^{(l)}$
$a_{D^{(l)}-1}^{(l)}$
$w_{(D^{(l+1)}-1)(D^{(l)}-1)}^{(l)}$
$z_0^{(l+1)} \rightarrow a_0^{(l+1)}$
$z_{D^{(l+1)}-1}^{(l+1)} \rightarrow a_{D^{(l+1)}-1}^{(l+1)}$

Figure 2.2: An affine layer and subsequent nonlinear function.

2.3 CONVOLUTION

Convolution kernels (commonly called filters) are widely adopted in computer vision and used with 2D images, 3D volumetric images, such as MRI scans, and 3D temporal images or video. Tasks where there is a correlation associated with the spatial or temporal proximity in the input values, such as in images, video, and spectrograms (discussed in Section 2.4), can use convolution kernels.

The term *convolution* has different meanings in the DL and signal processing literature. A convolution operator in the DL literature, and the one used in this book, is a *cross-correlation* operator between the weights and input activations (the input values to the convolutional layer). Each convolution output value is a dot product of the filter and a subset of the input. The entire convolution output is computed by shifting this filter across all the subsets in the input.

A 1D convolution using one filter follows:

$$z_i^{(l+1)} = \sum_{h=0}^{H-1} a_{h+i}^{(l)} w_h^{(l)} + b_i^{(l)},$$

where H is the length of filter $w^{(l)}$. This equation can be easily extended to a 2D convolution, which is more common in DL. Typically, multiple filters are used in each layer. Figure 2.3 illustrates K 1D convolutions and K 2D convolutions (the biases are omitted to simplify the figure).

The output is smaller if the input is not padded or if there is a stride between each filter shift. It is a common practice to extend or pad the input with zeros to enforce that the output size matches the input size (assuming the stride is 1). Another padding technique is using partial convolution, which generates a more fitting padding and is discussed elsewhere [LRS+18].

To demonstrate a 2D convolution, assume, for illustration purposes, a 6×6 gray-scaled input tensor (in practice, the input is usually much bigger) and a 5×5 filter, as shown in Fig-

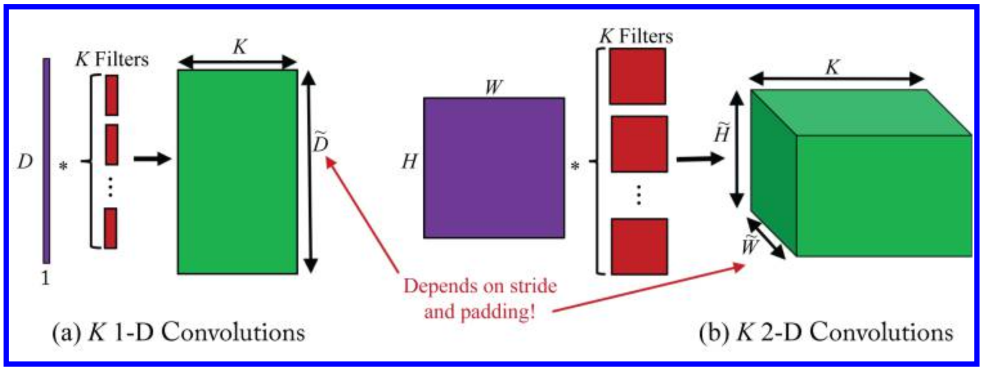
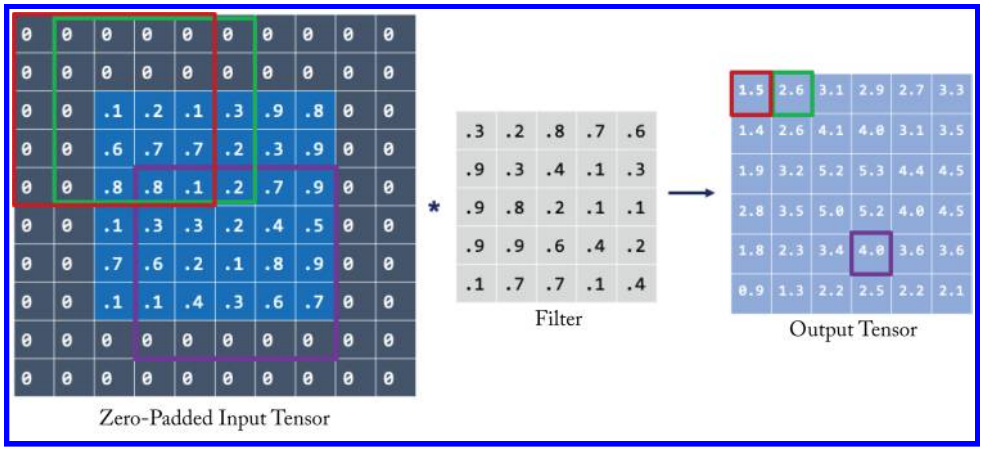


Figure 2.3: (a) K 1D convolutions and (b) K 2D convolutions. The results across all filters are concatenated across a new dimension. Thus, the output tensor of the K 2D convolutions has a depth (number of channels) of K.



0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	.1	.2	.1	.3	.9	.8	0	0
0	0	.6	.7	.7	.2	.3	.9	0	0
0	0	.8	.8	.1	.2	.7	.9	0	0
0	0	.1	.3	.3	.2	.4	.5	0	0
0	0	.7	.6	.2	.1	.8	.9	0	0
0	0	.1	.1	.4	.3	.6	.7	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

.3	.2	.8	.7	.6
.9	.3	.4	.1	.3
.9	.8	.2	.1	.1
.9	.9	.6	.4	.2
.1	.7	.7	.1	.4

1.5	2.6	3.1	2.9	2.7	3.3
1.4	2.6	4.1	4.0	3.1	3.5
1.9	3.2	5.2	5.3	4.4	4.5
2.8	3.5	5.0	5.2	4.0	4.5
1.8	2.3	3.4	4.0	3.6	3.6
0.9	1.3	2.2	2.5	2.2	2.1

Figure 2.4: A 2D convolution operation. The top-left value in the output tensor (right) is the dot product of the values in the filter (center) with the upper left values in input tensor (left) in the red square. The input tensor is first zero-padded so the output tensor height and width dimensions equal those of the input tensor. Credit: Joanne Yuan.

ure 2.4. The input is padded with zeros to ensure the output size equals the input size. The upper left value of the 2D output array is the dot product of the 5×5 filter with the upper-left 5×5 pixels in the zero-padded input tensor (marked in red). Note that in this book and the DL literature, the dots product's definition includes the aggregated sum of the Hadamard product (element-wise multiplication) of two 2D arrays. The next output value is computed using the next 5×5 values in the input tensor (marked in green). This pattern continues across the entire input array to compute all the output values.

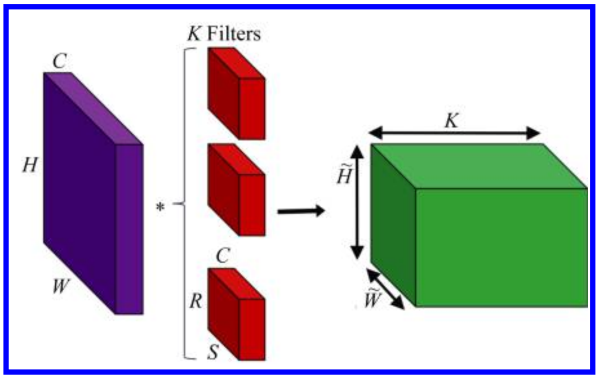


Figure 2.5: K 2D convolutions with an $H \times W$ input with C channels. Each filter also has C channels. The output tensor has K channels, each one corresponding to the convolution output of each filter with the input tensor.

An $H \times W$ color image has 3 channels (red, green, blue), also known as feature channels or tensor depth. The dimension of the image is represented as $3 \times H \times W$. The filters have the same number of channels as the input, as illustrated in Figure 2.5. Assuming K 5×5 filters with 3 channels (represented as $3 \times 5 \times 5$), each one of the K 2D convolutions is the dot product between a $3 \times 5 \times 5$ filter and all the $3 \times 5 \times 5$ subsets of the input shifted across the height and width. In 2D convolution, the filters do not shift across the depth (channels). Note that filter sizes are often described only by their height and width; the depth is inferred: it is the number of channels of the input tensor.

A convolutional layer has a bank of filters, each detecting different features in the input. To illustrate, suppose the input is a $3 \times 224 \times 224$ tensor, and the layer has a bank of 64 filters. Each filter produces one 224×224 output. Each output contains the features detected in the input by the corresponding filter. The aggregated layer output is a $64 \times 224 \times 224$ tensor, and all the filters in the next convolutional layer have 64 channels.

In practice, a convolution layer typically uses 4D input, filter, and output tensors. The usual way tensors are arranged in (1D) memory, known as the *data layout*, is as *NCHW* or *NHWC* for the input tensors, where N is the number of samples in the batch, C is the input depth (or equivalently, the number of channels or features), W is the input width, and H is the input height. The filters are arranged as *RSCK* or *KCRS*, where K is the number of filters (also known as the number of output feature channels), R is the filter height, and S is the filter width. The C in *NCWH* and *KCRS* are the same. Note that *KCRS* is sometimes denoted as *OIHW* in some literature but not in this book to avoid confusion with the H and W used for the input tensor. In the example above, the input has *NCHW* dimensions $1 \times 3 \times 224 \times 224$, the filter has *KCRS* dimensions $64 \times 3 \times 5 \times 5$, and the output has $NK\tilde{H}\tilde{W}$ dimensions $1 \times 64 \times 224 \times 224$.

The convolution is computed along seven dimensions: batch size N, output channels K, input channels C, output height $\tilde{H}$, output width $\tilde{W}$, filter height R, and filter width S. It can be implemented naively as seven `for` loops, as shown in Algorithm 2.1, where k, $\tilde{h}$, and $\tilde{w}$, represent the channel, height, and width indices of the output tensor $\mathbf{Z}$. For simplicity, the stride is assumed to be 1. There are more efficient implementations that account for a device's memory hierarchy and parallel computing capabilities [DAM+16].

Algorithm 2.1 Convolution primitive (Naive implementation)

```
init $\mathbf{z}^{(l+1)} = \mathbf{0}$
for $n \in 0, \cdots, N-1$ do
    for $k \in 0, \cdots, K-1$ do
        for $c \in 0, \cdots, C-1$ do
            for $\tilde{h} \in 0, \cdots, \tilde{H}-1$ do
                for $\tilde{w} \in 0, \cdots, \tilde{W}-1$ do
                    for $r \in 0, \cdots, R-1$ do
                        for $s \in 0, \cdots, S-1$ do
                            $z^{(l+1)}_{n,k,\tilde{h},\tilde{w}} \mathrel{+}= a^{(l)}_{n,c,\tilde{h}+r-1,\tilde{w}+s-1} \cdot w^{(l)}_{k,c,r,s}$
```

A convolutional filter can be implemented as a GEMM by duplicating some of the input values, as shown in Figure 2.6, and converting the filter into a vector. This is called an *im2col*-based convolution implementation. This implementation enables the use of a GEMM library (which is typically well optimized). However, it comes at the expense of additional memory requirements for the input tensor and extra compute cycles to transform the input. Conversely, an affine layer can be represented as a convolution layer where C is the number of input activations, K is the number of output activations, $H = W = 1$, and $R = S = 1$ using the *NCHW* and *KCRS* data layout representations.

In addition, a convolution operation can be implemented in the Winograd domain and also in Fourier domain using the Fast Fourier transform (FFT) algorithm [HR15, LG16]. oneDNN and cuDNN support Winograd convolutions, which may be beneficial for 3×3 convolutions (the maximum theoretical gain is $2.25\times$ over regular convolutions). However, using the Winograd transform may reduce the numeric accuracy of the computations, which can impact the overall accuracy of the model [LG16]. Sparsifying the Winograd domain (increasing the number of zero values in the Winograd domain) can lead to higher gains, but also higher accuracy loss [LPH+18]. Winograd requires additional memory and consumes more bandwidth; thus, when sufficient compute is available, the conversion overhead may result in overall slower performance. The FFT primarily benefits large filter sizes, which are uncommon in DL. Therefore, Winograd- and FFT-based convolutions are rarely used in production.

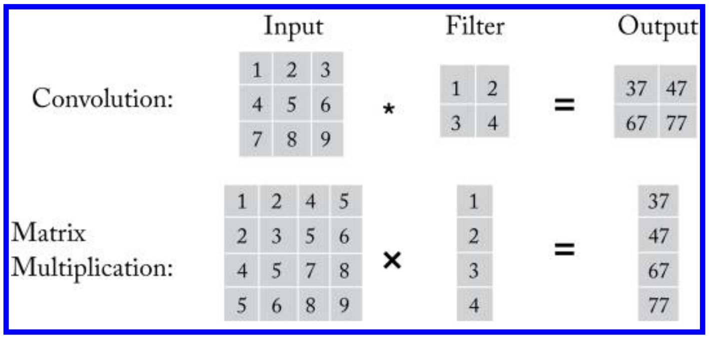


Figure 2.6: A convolution operation can be implemented as a matrix multiplication. In this simple illustration, the input is not zero-padded so the output dimensions are smaller than the input dimensions.

Section 3.2 introduces other variants of convolution, including the 1×1 convolution, group convolution, and depthwise separable convolutions, when discussing influential computer vision topologies.

2.4 POOLING

Pooling or subsampling reduces the size of the input tensor across the height and width, typically without affecting the number of channels. Pooling often follows a convolutional layer. The common implementation, known as *max pooling*, is to select the maximum value in a small region. A 2D pooling layer uses 2×2 nonoverlapping regions and reduces the tensor size by a factor of 4, as illustrated in Figure 2.7.

The main benefit of pooling is that filters after a pooling layer have a larger receptive field or coverage on the original input image. For example, a 3×3 filter maps to a 6×6 portion of the input image after one 2×2 pooling layer. A 3×3 filter deeper in the model, after five convolutional and pooling layers, maps to a 96×96 (note that $3 \times 2^5 = 96$) portion of the input image and can learn more complex features. Another benefit of pooling is that it reduces the number of operations.

Other forms of pooling include *average pooling*, *stochastic pooling*, and *spatial pyramid pooling* (SPP). Average pooling and stochastic pooling are similar to max pooling. Average pooling computes the average of the values in a small region. Stochastic pooling samples a value based on the distribution of the values in the small region [ZF13]. SPP is used after the last convolution layer to generate fixed-length outputs when the input images are not of a fixed size [HZR+15]. In Section 3.2.3, we provide an example of SPP used in a production model for image segmentation.

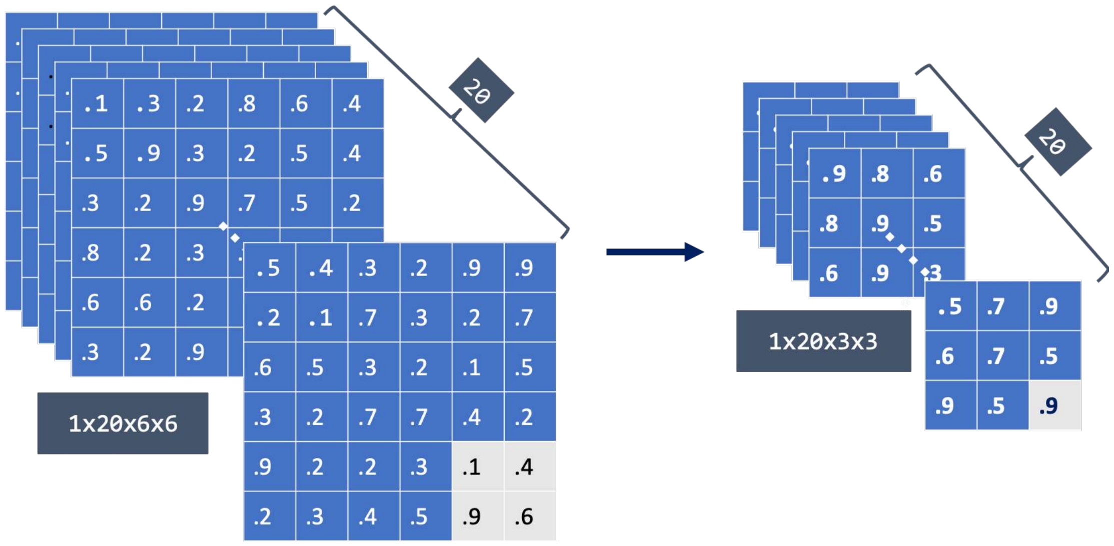


Figure 2.7: A (left) $1 \times 20 \times 6 \times 6$ tensor (in *NCHW* layout) input into a 2×2 pooling layer produces a (right) $1 \times 20 \times 3 \times 3$ tensor output. Credit: Joanne Yuan.

2.5 RECURRENT UNITS

There are three main types of recurrent units: vanilla RNN, Long Short Term Memory (LSTM), and Gated Recurrent Unit (GRU) [GSK+17, CGC+14]. Each unit has an internal vector or cell state, sometimes called the memory (not to be confused with a processor's memory). At every timestep, the input may modify this memory.

LSTM and GRU units have soft gates that control how the internal cell state is modified, as shown in Figure 2.8. These gates enable a NN to retain information for several timesteps. LSTM units have the most extensive adoption, comparable performance to GRU units, and typically statistically outperform vanilla RNN units.

LSTM and GRU units do not use activation functions between recurrent components. Therefore, the gradient does not tend to vanish during backpropagation. An LSTM and a GRU unit contain gates that allow them to control the information flow. An LSTM has a "forget" gate to flush memory cell's values, an "input" gate to add new inputs to the memory cell, and an "output" gate to get values from the memory cell. Multiplying the gate input value with the output value of a sigmoid function (the gate), which corresponds to a number between 0 and 1, implements this gating. Note the input, output, and memory cell are vectors, and each vector's value uses a unique gating value.