Dive into Deep Learning

Release 0.14.4

Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola

Sep 18, 2020

Contents

Preface

Just a few years ago, there were no legions of deep learning scientists developing intelligent products and services at major companies and startups. When the youngest among us (the authors) entered the field, machine learning did not command headlines in daily newspapers. Our parents had no idea what machine learning was, let alone why we might prefer it to a career in medicine or law. Machine learning was a forward-looking academic discipline with a narrow set of real-world applications. And those applications, e.g., speech recognition and computer vision, required so much domain knowledge that they were often regarded as separate areas entirely for which machine learning was one small component. Neural networks then, the antecedents of the deep learning models that we focus on in this book, were regarded as outmoded tools.

In just the past five years, deep learning has taken the world by surprise, driving rapid progress in fields as diverse as computer vision, natural language processing, automatic speech recognition, reinforcement learning, and statistical modeling. With these advances in hand, we can now build cars that drive themselves with more autonomy than ever before (and less autonomy than some companies might have you believe), smart reply systems that automatically draft the most mundane emails, helping people dig out from oppressively large inboxes, and software agents that dominate the world's best humans at board games like Go, a feat once thought to be decades away. Already, these tools exert ever-wider impacts on industry and society, changing the way movies are made, diseases are diagnosed, and playing a growing role in basic sciences—from astrophysics to biology.

About This Book

This book represents our attempt to make deep learning approachable, teaching you the *concepts*, the *context*, and the *code*.

One Medium Combining Code, Math, and HTML

For any computing technology to reach its full impact, it must be well-understood, well-documented, and supported by mature, well-maintained tools. The key ideas should be clearly distilled, minimizing the onboarding time needing to bring new practitioners up to date. Mature libraries should automate common tasks, and exemplar code should make it easy for practitioners to modify, apply, and extend common applications to suit their needs. Take dynamic web applications as an example. Despite a large number of companies, like Amazon, developing successful database-driven web applications in the 1990s, the potential of this technology to aid creative entrepreneurs has been realized to a far greater degree in the past ten years, owing in part to the development of powerful, well-documented frameworks.

Testing the potential of deep learning presents unique challenges because any single application brings together various disciplines. Applying deep learning requires simultaneously understanding (i) the motivations for casting a problem in a particular way; (ii) the mathematics of a given modeling approach; (iii) the optimization algorithms for fitting the models to data; and (iv) the engineering required to train models efficiently, navigating the pitfalls of numerical computing and getting the most out of available hardware. Teaching both the critical thinking skills required to formulate problems, the mathematics to solve them, and the software tools to implement those solutions all in one place presents formidable challenges. Our goal in this book is to present a unified resource to bring would-be practitioners up to speed.

At the time we started this book project, there were no resources that simultaneously (i) were up to date; (ii) covered the full breadth of modern machine learning with substantial technical depth; and (iii) interleaved exposition of the quality one expects from an engaging textbook with the clean runnable code that one expects to find in hands-on tutorials. We found plenty of code examples for how to use a given deep learning framework (e.g., how to do basic numerical computing with matrices in TensorFlow) or for implementing particular techniques (e.g., code snippets for LeNet, AlexNet, ResNets, etc) scattered across various blog posts and GitHub repositories. However, these examples typically focused on *how* to implement a given approach, but left out the discussion of *why* certain algorithmic decisions are made. While some interactive resources have popped up sporadically to address a particular topic, e.g., the engaging blog posts published on the website Distill[3], or personal blogs, they only covered selected topics in deep learning, and often lacked associated code. On the other hand, while several textbooks have emerged, most notably (Goodfellow et al., 2016), which offers a comprehensive survey of the concepts behind deep learning, these resources do not marry the descriptions to realizations of the concepts in code, sometimes leaving readers clueless as to how to implement them. Moreover, too many resources are hidden behind the paywalls of commercial course providers.

We set out to create a resource that could (i) be freely available for everyone; (ii) offer sufficient technical depth to provide a starting point on the path to actually becoming an applied machine learning scientist; (iii) include runnable code, showing readers *how* to solve problems in practice; (iv) allow for rapid updates, both by us and also by the community at large; and (v) be complemented by a forum[4] for interactive discussion of technical details and to answer questions.

These goals were often in conflict. Equations, theorems, and citations are best managed and laid out in LaTeX. Code is best described in Python. And webpages are native in HTML and JavaScript. Furthermore, we want the content to be accessible both as executable code, as a physical book, as a downloadable PDF, and on the Internet as a website. At present there exist no tools and no workflow perfectly suited to these demands, so we had to assemble our own. We describe our approach in detail in Section 19.6. We settled on GitHub to share the source and to allow for edits, Jupyter notebooks for mixing code, equations and text, Sphinx as a rendering engine to generate multiple outputs, and Discourse for the forum. While our system is not yet perfect, these choices provide a good compromise among the competing concerns. We believe that this might be the first book published using such an integrated workflow.

[3] http://distill.pub
[4] http://discuss.d2l.ai

Learning by Doing

Many textbooks teach a series of topics, each in exhaustive detail. For example, Chris Bishop's excellent textbook (Bishop, 2006), teaches each topic so thoroughly, that getting to the chapter on linear regression requires a non-trivial amount of work. While experts love this book precisely for its thoroughness, for beginners, this property limits its usefulness as an introductory text.

In this book, we will teach most concepts *just in time*. In other words, you will learn concepts at the very moment that they are needed to accomplish some practical end. While we take some time at the outset to teach fundamental preliminaries, like linear algebra and probability, we want you to taste the satisfaction of training your first model before worrying about more esoteric probability distributions.

Aside from a few preliminary notebooks that provide a crash course in the basic mathematical background, each subsequent chapter introduces both a reasonable number of new concepts and provides single self-contained working examples—using real datasets. This presents an organizational challenge. Some models might logically be grouped together in a single notebook. And some ideas might be best taught by executing several models in succession. On the other hand, there is a big advantage to adhering to a policy of *1 working example, 1 notebook*: This makes it as easy as possible for you to start your own research projects by leveraging our code. Just copy a notebook and start modifying it.

We will interleave the runnable code with background material as needed. In general, we will often err on the side of making tools available before explaining them fully (and we will follow up by explaining the background later). For instance, we might use *stochastic gradient descent* before fully explaining why it is useful or why it works. This helps to give practitioners the necessary ammunition to solve problems quickly, at the expense of requiring the reader to trust us with some curatorial decisions.

This book will teach deep learning concepts from scratch. Sometimes, we want to delve into fine details about the models that would typically be hidden from the user by deep learning frameworks' advanced abstractions. This comes up especially in the basic tutorials, where we want you to understand everything that happens in a given layer or optimizer. In these cases, we will often present two versions of the example: one where we implement everything from scratch, relying only on the NumPy interface and automatic differentiation, and another, more practical example, where we write succinct code using Gluon. Once we have taught you how some component works, we can just use the Gluon version in subsequent tutorials.

Content and Structure

The book can be roughly divided into three parts, which are presented by different colors in Fig. 1:

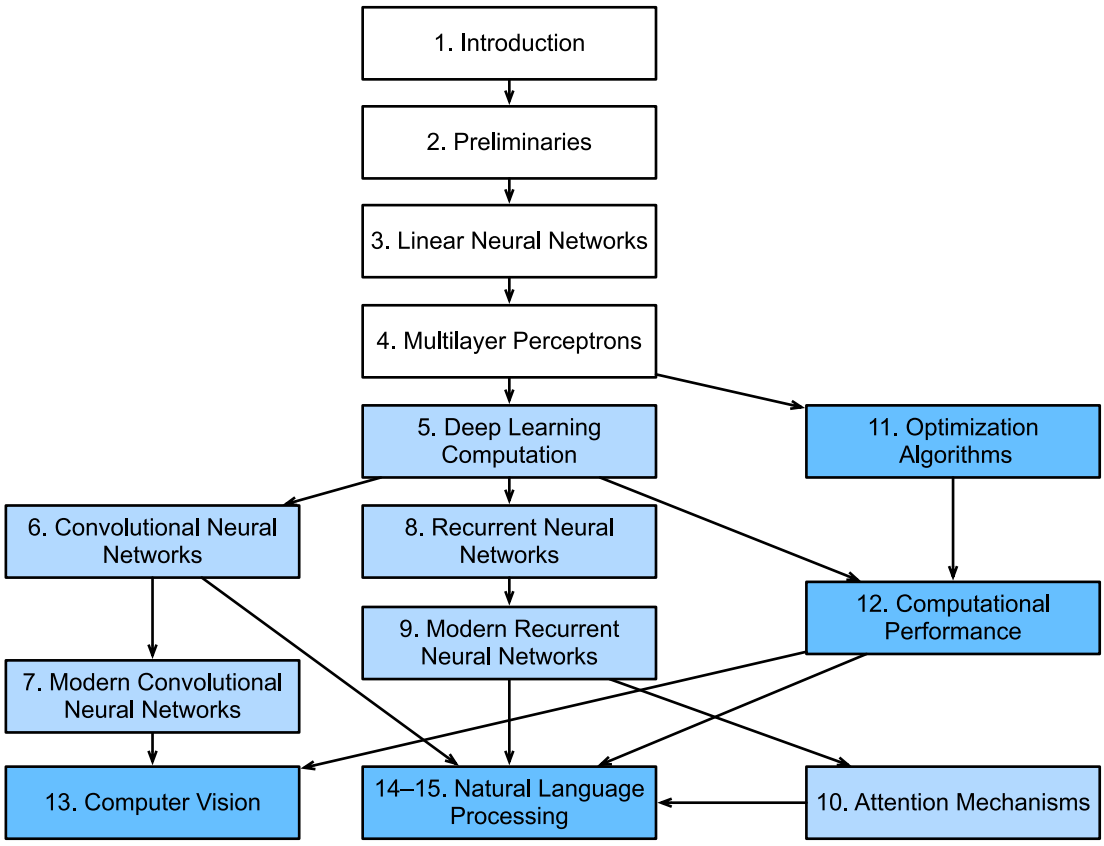


Fig. 1: Book structure

- The first part covers basics and preliminaries. Chapter 1 offers an introduction to deep learning. Then, in Chapter 2, we quickly bring you up to speed on the prerequisites required for hands-on deep learning, such as how to store and manipulate data, and how to apply various numerical operations based on basic concepts from linear algebra, calculus, and probability. Chapter 3 and Chapter 4 cover the most basic concepts and techniques of deep learning, such as linear regression, multilayer perceptrons and regularization.
- The next five chapters focus on modern deep learning techniques. Chapter 5 describes the various key components of deep learning calculations and lays the groundwork for us to subsequently implement more complex models. Next, in Chapter 6 and Chapter 7, we introduce convolutional neural networks (CNNs), powerful tools that form the backbone of most modern computer vision systems. Subsequently, in Chapter 8 and Chapter 9, we introduce recurrent neural networks (RNNs), models that exploit temporal or sequential structure in data, and are commonly used for natural language processing and time series prediction. In Chapter 10, we introduce a new class of models that employ a technique called attention mechanisms and they have recently begun to displace RNNs in natural language processing. These sections will get you up to speed on the basic tools behind most modern applications of deep learning.
- Part three discusses scalability, efficiency, and applications. First, in Chapter 11, we discuss several common optimization algorithms used to train deep learning models. The next

chapter, Chapter 12 examines several key factors that influence the computational performance of your deep learning code. In Chapter 13, we illustrate major applications of deep learning in computer vision. In Chapter 14 and Chapter 15, we show how to pretrain language representation models and apply them to natural language processing tasks.

Code

Most sections of this book feature executable code because of our belief in the importance of an interactive learning experience in deep learning. At present, certain intuitions can only be developed through trial and error, tweaking the code in small ways and observing the results. Ideally, an elegant mathematical theory might tell us precisely how to tweak our code to achieve a desired result. Unfortunately, at present, such elegant theories elude us. Despite our best attempts, formal explanations for various techniques are still lacking, both because the mathematics to characterize these models can be so difficult and also because serious inquiry on these topics has only just recently kicked into high gear. We are hopeful that as the theory of deep learning progresses, future editions of this book will be able to provide insights in places the present edition cannot.

At times, to avoid unnecessary repetition, we encapsulate the frequently-imported and referred-to functions, classes, etc. in this book in the `d2l` package. For any block such as a function, a class, or multiple imports to be saved in the package, we will mark it with `#@save`. We offer a detailed overview of these functions and classes in Section 19.7. The `d2l` package is light-weight and only requires the following packages and modules as dependencies:

```
#@save
import collections
from collections import defaultdict
from IPython import display
import math
from matplotlib import pyplot as plt
import os
import pandas as pd
import random
import re
import shutil
import sys
import tarfile
import time
import requests
import zipfile
import hashlib
d2l = sys.modules[__name__]
```

Most of the code in this book is based on Apache MXNet. MXNet is an open-source framework for deep learning and the preferred choice of AWS (Amazon Web Services), as well as many colleges and companies. All of the code in this book has passed tests under the newest MXNet version. However, due to the rapid development of deep learning, some code *in the print edition* may not work properly in future versions of MXNet. However, we plan to keep the online version up-to-date. In case you encounter any such problems, please consult *Installation* (page 9) to update your code and runtime environment.

Here is how we import modules from MXNet.

```
#@save
from mxnet import autograd, context, gluon, image, init, np, npx
from mxnet.gluon import nn, rnn
```

Target Audience

This book is for students (undergraduate or graduate), engineers, and researchers, who seek a solid grasp of the practical techniques of deep learning. Because we explain every concept from scratch, no previous background in deep learning or machine learning is required. Fully explaining the methods of deep learning requires some mathematics and programming, but we will only assume that you come in with some basics, including (the very basics of) linear algebra, calculus, probability, and Python programming. Moreover, in the Appendix, we provide a refresher on most of the mathematics covered in this book. Most of the time, we will prioritize intuition and ideas over mathematical rigor. There are many terrific books which can lead the interested reader further. For instance, Linear Analysis by Bela Bollobas (Bollobas, 1999) covers linear algebra and functional analysis in great depth. All of Statistics (Wasserman, 2013) is a terrific guide to statistics. And if you have not used Python before, you may want to peruse this Python tutorial[5].

Forum

Associated with this book, we have launched a discussion forum, located at discuss.d2l.ai[6]. When you have questions on any section of the book, you can find the associated discussion page link at the end of each chapter.

Acknowledgments

We are indebted to the hundreds of contributors for both the English and the Chinese drafts. They helped improve the content and offered valuable feedback. Specifically, we thank every contributor of this English draft for making it better for everyone. Their GitHub IDs or names are (in no particular order): alxnorden, avinashingit, bowen0701, brettkoonce, Chaitanya Prakash Bapat, cryptonaut, Davide Fiocco, edgarroman, gkutiel, John Mitro, Liang Pu, Rahul Agarwal, Mohamed Ali Jamaoui, Michael (Stu) Stewart, Mike Müller, NRauschmayr, Prakhar Srivastav, sad-, sfermigier, Sheng Zha, sundeepteki, topecongiro, tpdi, vermicelli, Vishaal Kapoor, Vishwesh Ravi Shrimali, YaYaB, Yuhong Chen, Evgeniy Smirnov, lgov, Simon Corston-Oliver, Igor Dzreyev, Ha Nguyen, pmuens, Andrei Lukovenko, senorcinco, vfdev-5, dsweet, Mohammad Mahdi Rahimi, Abhishek Gupta, uwsd, DomKM, Lisa Oakley, Bowen Li, Aarush Ahuja, Prasanth Buddareddygari, brianhendee, mani2106, mtn, lkevinzc, caojilin, Lakshya, Fiete Lüer, Surbhi Vijayvargeeya, Muhyun Kim, dennismalmgren, adursun, Anirudh Dagar, liqingnz, Pedro Larroy, lgov, ati-ozgur, Jun Wu, Matthias Blume, Lin Yuan, geogunow, Josh Gardner, Maximilian Böther, Rakib Islam, Leonard Lausen, Abhinav Upadhyay, rongruosong, Steve Sedlmeyer, Ruslan Baratov, Rafael Schlatter, liusy182, Giannis Pappas, ati-ozgur, qbaza, dchoi77, Adam Gerson, Phuc Le, Mark Atwood, christabella, vn09, Haibin Lin, jjangga0214, RichyChen, noelo, hansent, Giel Dops, dvincent1337, WhiteD3vil, Peter Kulits, codypenta, joseppinilla, ahmaurya, karolszk,

[5] http://learnpython.org/
[6] https://discuss.d2l.ai/

heytitle, Peter Goetz, rigtorp, tiepvupsu, sfilip, mlxd, Kale-ab Tessera, Sanjar Adilov, Matteo-Ferrara, hsneto, Katarzyna Biesialska, Gregory Bruss, duythanhvn, paulaurel, graytowne, minhduc0711, sl7423, Jaedong Hwang, Yida Wang, cys4, clhm, Jean Kaddour, austinmw, trebeljahr, tbaums, cuongvng, pavelkomarov, vzlamal, NotAnotherSystem, J-Arun-Mani, jancio, eldarkurtic, the-great-shazbot, doctorcolossus, gducharme, cclauss, Daniel-Mietchen, hoonose, biagiom, abhinavsp0730, jonathanhrandall, ysraell, Nodar Okroshiashvili, UgurKap, Jiyang Kang, Steven-Jokes, Tomer Kaftan, liweiwp, netyster, ypandya, NishantTharani, heiligerl, SportsTHU, nguyenhoa93, manuel-arno-korfmann-webentwicklung, aterzis-personal, nxby, Xiaoting He, Josiah Yoder, mathresearch, mzz2017, jroberayalas, iluu, ghejc, BSharmi, vkramdev, simonwardjones, LakshKD, TalNeoran.

We thank Amazon Web Services, especially Swami Sivasubramanian, Raju Gulabani, Charlie Bell, and Andrew Jassy for their generous support in writing this book. Without the available time, resources, discussions with colleagues, and continuous encouragement this book would not have happened.

Summary

- Deep learning has revolutionized pattern recognition, introducing technology that now powers a wide range of technologies, including computer vision, natural language processing, automatic speech recognition.
- To successfully apply deep learning, you must understand how to cast a problem, the mathematics of modeling, the algorithms for fitting your models to data, and the engineering techniques to implement it all.
- This book presents a comprehensive resource, including prose, figures, mathematics, and code, all in one place.
- To answer questions related to this book, visit our forum at https://discuss.d2l.ai/.
- All notebooks are available for download on GitHub.

Exercises

1. Register an account on the discussion forum of this book discuss.d2l.ai[7].
2. Install Python on your computer.
3. Follow the links at the bottom of the section to the forum, where you will be able to seek out help and discuss the book and find answers to your questions by engaging the authors and broader community.

Discussions[8]

[7] https://discuss.d2l.ai/
[8] https://discuss.d2l.ai/t/18

Installation

In order to get you up and running for hands-on learning experience, we need to set you up with an environment for running Python, Jupyter notebooks, the relevant libraries, and the code needed to run the book itself.

Installing Miniconda

The simplest way to get going will be to install Miniconda[9]. The Python 3.x version is required. You can skip the following steps if conda has already been installed. Download the corresponding Miniconda sh file from the website and then execute the installation from the command line using `sh <FILENAME> -b`. For macOS users:

```
# The file name is subject to changes
sh Miniconda3-latest-MacOSX-x86_64.sh -b
```

For Linux users:

```
# The file name is subject to changes
sh Miniconda3-latest-Linux-x86_64.sh -b
```

Next, initialize the shell so we can run `conda` directly.

```
~/miniconda3/bin/conda init
```

Now close and re-open your current shell. You should be able to create a new environment as following:

```
conda create --name d2l -y
```

[9] https://conda.io/en/latest/miniconda.html

Downloading the D2L Notebooks

Next, we need to download the code of this book. You can click the "All Notebooks" tab on the top of any HTML page to download and unzip the code. Alternatively, if you have `unzip` (otherwise run `sudo apt install unzip`) available:

```
mkdir d2l-en && cd d2l-en
curl https://d2l.ai/d2l-en.zip -o d2l-en.zip
unzip d2l-en.zip && rm d2l-en.zip
```

Now we will want to activate the `d2l` environment and install `pip`. Enter `y` for the queries that follow this command.

```
conda activate d2l
conda install pip -y
```

Installing the Framework and the `d2l` Package

Before installing the deep learning framework, please first check whether or not you have proper GPUs on your machine (the GPUs that power the display on a standard laptop do not count for our purposes). If you are installing on a GPU server, proceed to *GPU Support* (page 11) for instructions to install a GPU-supported version.

Otherwise, you can install the CPU version as follows. That will be more than enough horsepower to get you through the first few chapters but you will want to access GPUs before running larger models.

```
pip install mxnet==1.7.0
```

We also install the `d2l` package that encapsulates frequently used functions and classes in this book.

```
# -U: Upgrade all packages to the newest available version
pip install -U d2l
```

Once they are installed, we now open the Jupyter notebook by running:

```
jupyter notebook
```

At this point, you can open http://localhost:8888 (it usually opens automatically) in your Web browser. Then we can run the code for each section of the book. Please always execute `conda activate d2l` to activate the runtime environment before running the code of the book or updating the deep learning framework or the `d2l` package. To exit the environment, run `conda deactivate`.

GPU Support

By default, MXNet is installed without GPU support to ensure that it will run on any computer (including most laptops). Part of this book requires or recommends running with GPU. If your computer has NVIDIA graphics cards and has installed CUDA[10], then you should install a GPU-enabled version. If you have installed the CPU-only version, you may need to remove it first by running:

```
pip uninstall mxnet
```

Then we need to find the CUDA version you installed. You may check it through `nvcc --version` or `cat /usr/local/cuda/version.txt`. Assume that you have installed CUDA 10.1, then you can install with the following command:

```
# For Windows users
pip install mxnet-cu101==1.7.0 -f https://dist.mxnet.io/python

# For Linux and macOS users
pip install mxnet-cu101==1.7.0
```

You may change the last digits according to your CUDA version, e.g., `cu100` for CUDA 10.0 and `cu90` for CUDA 9.0.

Exercises

1. Download the code for the book and install the runtime environment.

Discussions[11]

[10] https://developer.nvidia.com/cuda-downloads
[11] https://discuss.d2l.ai/t/23

Notation

The notation used throughout this book is summarized below.

Numbers

- x: A scalar
- $\mathbf{x}$: A vector
- $\mathbf{X}$: A matrix
- X: A tensor
- $\mathbf{I}$: An identity matrix
- x_i, $[\mathbf{x}]_i$: The i^{th} element of vector $\mathbf{x}$
- x_{ij}, $x_{i,j}$,$[\mathbf{X}]_{ij}$, $[\mathbf{X}]_{i,j}$: The element of matrix $\mathbf{X}$ at row i and column j

Set Theory

- $\mathcal{X}$: A set
- $\mathbb{Z}$: The set of integers
- $\mathbb{Z}^+$: The set of positive integers
- $\mathbb{R}$: The set of real numbers
- $\mathbb{R}^n$: The set of n-dimensional vectors of real numbers
- $\mathbb{R}^{a\times b}$: The set of matrices of real numbers with a rows and b columns
- $|\mathcal{X}|$: Cardinality (number of elements) of set $\mathcal{X}$
- $\mathcal{A} \cup \mathcal{B}$: Union of sets $\mathcal{A}$ and $\mathcal{B}$
- $\mathcal{A} \cap \mathcal{B}$: Intersection of sets $\mathcal{A}$ and $\mathcal{B}$
- $\mathcal{A} \setminus \mathcal{B}$: Subtraction of set $\mathcal{B}$ from set $\mathcal{A}$

Functions and Operators

- $f(\cdot)$: A function
- $\log(\cdot)$: The natural logarithm
- $\exp(\cdot)$: The exponential function
- $\mathbf{1}_{\mathcal{X}}$: The indicator function
- $(\cdot)^\top$: Transpose of a vector or a matrix
- $\mathbf{X}^{-1}$: Inverse of matrix $\mathbf{X}$
- $\odot$: Hadamard (elementwise) product
- $[\cdot, \cdot]$: Concatenation
- $|\mathcal{X}|$: Cardinality of set $\mathcal{X}$
- $\|\cdot\|_p$: L_p norm
- $\|\cdot\|$: L_2 norm
- $\langle \mathbf{x}, \mathbf{y} \rangle$: Dot product of vectors $\mathbf{x}$ and $\mathbf{y}$
- $\sum$: Series addition
- $\prod$: Series multiplication
- $\stackrel{\text{def}}{=}$: Definition

Calculus

- $\frac{dy}{dx}$: Derivative of y with respect to x
- $\frac{\partial y}{\partial x}$: Partial derivative of y with respect to x
- $\nabla_{\mathbf{x}} y$: Gradient of y with respect to $\mathbf{x}$
- $\int_a^b f(x)\, dx$: Definite integral of f from a to b with respect to x
- $\int f(x)\, dx$: Indefinite integral of f with respect to x

Probability and Information Theory

- $P(\cdot)$: Probability distribution
- $z \sim P$: Random variable z has probability distribution P
- $P(X \mid Y)$: Conditional probability of $X \mid Y$
- $p(x)$: Probability density function
- $E_x[f(x)]$: Expectation of f with respect to x
- $X \perp Y$: Random variables X and Y are independent

- $X \perp Y \mid Z$: Random variables X and Y are conditionally independent given random variable Z
- $\mathrm{Var}(X)$: Variance of random variable X
- σ_X: Standard deviation of random variable X
- $\mathrm{Cov}(X, Y)$: Covariance of random variables X and Y
- $\rho(X, Y)$: Correlation of random variables X and Y
- $H(X)$: Entropy of random variable X
- $D_{\mathrm{KL}}(P\|Q)$: KL-divergence of distributions P and Q

Complexity

- $\mathcal{O}$: Big O notation

Discussions[12]

[12] https://discuss.d2l.ai/t/25

1 | Introduction

Until recently, nearly every computer program that we interact with daily was coded by software developers from first principles. Say that we wanted to write an application to manage an e-commerce platform. After huddling around a whiteboard for a few hours to ponder the problem, we would come up with the broad strokes of a working solution that might probably look something like this: (i) users interact with the application through an interface running in a web browser or mobile application; (ii) our application interacts with a commercial-grade database engine to keep track of each user's state and maintain records of historical transactions; and (iii) at the heart of our application, the *business logic* (you might say, the *brains*) of our application spells out in methodical detail the appropriate action that our program should take in every conceivable circumstance.

To build the *brains* of our application, we'd have to step through every possible corner case that we anticipate encountering, devising appropriate rules. Each time a customer clicks to add an item to their shopping cart, we add an entry to the shopping cart database table, associating that user's ID with the requested product's ID. While few developers ever get it completely right the first time (it might take some test runs to work out the kinks), for the most part, we could write such a program from first principles and confidently launch it *before ever seeing a real customer*. Our ability to design automated systems from first principles that drive functioning products and systems, often in novel situations, is a remarkable cognitive feat. And when you are able to devise solutions that work 100% of the time, *you should not be using machine learning*.

Fortunately for the growing community of machine learning (ML) scientists, many tasks that we would like to automate do not bend so easily to human ingenuity. Imagine huddling around the whiteboard with the smartest minds you know, but this time you are tackling one of the following problems:

- Write a program that predicts tomorrow's weather given geographic information, satellite images, and a trailing window of past weather.
- Write a program that takes in a question, expressed in free-form text, and answers it correctly.
- Write a program that given an image can identify all the people it contains, drawing outlines around each.
- Write a program that presents users with products that they are likely to enjoy but unlikely, in the natural course of browsing, to encounter.

In each of these cases, even elite programmers are incapable of coding up solutions from scratch. The reasons for this can vary. Sometimes the program that we are looking for follows a pattern that changes over time, and we need our programs to adapt. In other cases, the relationship (say between pixels, and abstract categories) may be too complicated, requiring thousands or millions of computations that are beyond our conscious understanding (even if our eyes manage the task

effortlessly). ML is the study of powerful techniques that can *learn* from *experience*. As an ML algorithm accumulates more experience, typically in the form of observational data or interactions with an environment, its performance improves. Contrast this with our deterministic e-commerce platform, which performs according to the same business logic, no matter how much experience accrues, until the developers themselves *learn* and decide that it is time to update the software. In this book, we will teach you the fundamentals of machine learning, and focus in particular on deep learning, a powerful set of techniques driving innovations in areas as diverse as computer vision, natural language processing, healthcare, and genomics.

1.1 A Motivating Example

Before we could begin writing, the authors of this book, like much of the work force, had to become caffeinated. We hopped in the car and started driving. Using an iPhone, Alex called out "Hey Siri", awakening the phone's voice recognition system. Then Mu commanded "directions to Blue Bottle coffee shop". The phone quickly displayed the transcription of his command. It also recognized that we were asking for directions and launched the Maps application to fulfill our request. Once launched, the Maps app identified a number of routes. Next to each route, the phone displayed a predicted transit time. While we fabricated this story for pedagogical convenience, it demonstrates that in the span of just a few seconds, our everyday interactions with a smart phone can engage several machine learning models.

Imagine just writing a program to respond to a *wake word* like "Alexa", "Okay, Google" or "Siri". Try coding it up in a room by yourself with nothing but a computer and a code editor, as illustrated in Fig. 1.1.1. How would you write such a program from first principles? Think about it... the problem is hard. Every second, the microphone will collect roughly 44,000 samples. Each sample is a measurement of the amplitude of the sound wave. What rule could map reliably from a snippet of raw audio to confident predictions `{yes, no}` on whether the snippet contains the wake word? If you are stuck, do not worry. We do not know how to write such a program from scratch either. That is why we use ML.

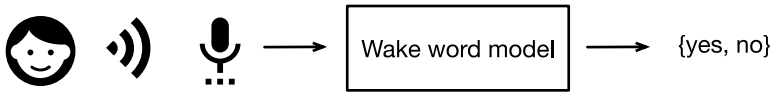


Fig. 1.1.1: Identify an awake word.

Here's the trick. Often, even when we do not know how to tell a computer explicitly how to map from inputs to outputs, we are nonetheless capable of performing the cognitive feat ourselves. In other words, even if you do not know *how to program a computer* to recognize the word "Alexa", you yourself *are able* to recognize the word "Alexa". Armed with this ability, we can collect a huge *dataset* containing examples of audio and label those that *do* and that *do not* contain the wake word. In the ML approach, we do not attempt to design a system *explicitly* to recognize wake words. Instead, we define a flexible program whose behavior is determined by a number of *parameters*. Then we use the dataset to determine the best possible set of parameters, those that improve the performance of our program with respect to some measure of performance on the task of interest.

You can think of the parameters as knobs that we can turn, manipulating the behavior of the program. Fixing the parameters, we call the program a *model*. The set of all distinct programs (input-output mappings) that we can produce just by manipulating the parameters is called a *family* of

models. And the *meta-program* that uses our dataset to choose the parameters is called a *learning algorithm*.

Before we can go ahead and engage the learning algorithm, we have to define the problem precisely, pinning down the exact nature of the inputs and outputs, and choosing an appropriate model family. In this case, our model receives a snippet of audio as *input*, and it generates a selection among `{yes, no}` as *output*. If all goes according to plan the model's guesses will typically be correct as to whether (or not) the snippet contains the wake word.

If we choose the right family of models, then there should exist one setting of the knobs such that the model fires `yes` every time it hears the word "Alexa". Because the exact choice of the wake word is arbitrary, we will probably need a model family sufficiently rich that, via another setting of the knobs, it could fire `yes` only upon hearing the word "Apricot". We expect that the same model family should be suitable for *"Alexa" recognition* and *"Apricot" recognition* because they seem, intuitively, to be similar tasks. However, we might need a different family of models entirely if we want to deal with fundamentally different inputs or outputs, say if we wanted to map from images to captions, or from English sentences to Chinese sentences.

As you might guess, if we just set all of the knobs randomly, it is not likely that our model will recognize "Alexa", "Apricot", or any other English word. In deep learning, the *learning* is the process by which we discover the right setting of the knobs coercing the desired behavior from our model.

As shown in Fig. 1.1.2, the training process usually looks like this:

1. Start off with a randomly initialized model that cannot do anything useful.
2. Grab some of your labeled data (e.g., audio snippets and corresponding `{yes, no}` labels)
3. Tweak the knobs so the model sucks less with respect to those examples
4. Repeat until the model is awesome.

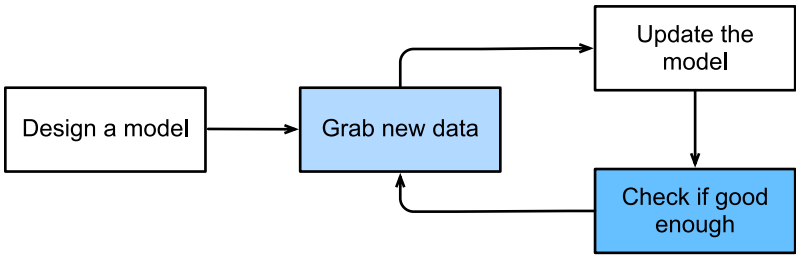
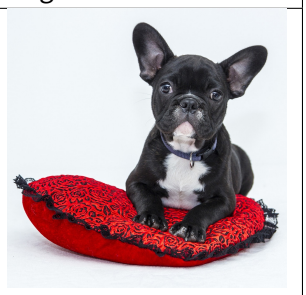


Fig. 1.1.2: A typical training process.

To summarize, rather than code up a wake word recognizer, we code up a program that can *learn* to recognize wake words, *if we present it with a large labeled dataset*. You can think of this act of determining a program's behavior by presenting it with a dataset as *programming with data*. We can "program" a cat detector by providing our machine learning system with many examples of cats and dogs, such as the images below:

cat	cat	dog	dog
			

This way the detector will eventually learn to emit a very large positive number if it is a cat, a very large negative number if it is a dog, and something closer to zero if it is not sure, and this barely scratches the surface of what ML can do.

Deep learning is just one among many popular methods for solving machine learning problems. Thus far, we have only talked about machine learning broadly and not deep learning. To see why deep learning is important, we should pause for a moment to highlight a couple of crucial points.

First, the problems that we have discussed thus far—learning from the raw audio signal, the raw pixel values of images, or mapping between sentences of arbitrary lengths and their counterparts in foreign languages—are problems where deep learning excels and where traditional ML methods faltered. Deep models are *deep* in precisely the sense that they learn many *layers* of computation. It turns out that these many-layered (or hierarchical) models are capable of addressing low-level perceptual data in a way that previous tools could not. In bygone days, the crucial part of applying ML to these problems consisted of coming up with manually-engineered ways of transforming the data into some form amenable to *shallow* models. One key advantage of deep learning is that it replaces not only the *shallow* models at the end of traditional learning pipelines, but also the labor-intensive process of feature engineering. Second, by replacing much of the *domain-specific preprocessing*, deep learning has eliminated many of the boundaries that previously separated computer vision, speech recognition, natural language processing, medical informatics, and other application areas, offering a unified set of tools for tackling diverse problems.

1.2 The Key Components: Data, Models, and Algorithms

In our *wake-word* example, we described a dataset consisting of audio snippets and binary labels, and we gave a hand-wavy sense of how we might *train* a model to approximate a mapping from snippets to classifications. This sort of problem, where we try to predict a designated unknown *label* given known *inputs*, given a dataset consisting of examples, for which the labels are known is called *supervised learning*, and it is just one among many *kinds* of machine learning problems. In the next section, we will take a deep dive into the different ML problems. First, we'd like to shed more light on some core components that will follow us around, no matter what kind of ML problem we take on:

1. The *data* that we can learn from.
2. A *model* of how to transform the data.
3. A *loss* function that quantifies the *badness* of our model.
4. An *algorithm* to adjust the model's parameters to minimize the loss.

1.2.1 Data

It might go without saying that you cannot do data science without data. We could lose hundreds of pages pondering what precisely constitutes data, but for now, we will err on the practical side and focus on the key properties to be concerned with. Generally, we are concerned with a collection of *examples*. In order to work with data usefully, we typically need to come up with a suitable numerical representation. Each *example* typically consists of a collection of numerical attributes called *features*. In the supervised learning problems above, a special feature is designated as the prediction *target*, (sometimes called the *label* or *dependent variable*). The given features from which the model must make its predictions can then simply be called the *features*, (or often, the *inputs*, *covariates*, or *independent variables*).

If we were working with image data, each individual photograph might constitute an *example*, each represented by an ordered list of numerical values corresponding to the brightness of each pixel. A 200×200 color photograph would consist of $200 \times 200 \times 3 = 120000$ numerical values, corresponding to the brightness of the red, green, and blue channels for each spatial location. In a more traditional task, we might try to predict whether or not a patient will survive, given a standard set of features such as age, vital signs, diagnoses, etc.

When every example is characterized by the same number of numerical values, we say that the data consists of *fixed-length* vectors and we describe the (constant) length of the vectors as the *dimensionality* of the data. As you might imagine, fixed-length can be a convenient property. If we wanted to train a model to recognize cancer in microscopy images, fixed-length inputs mean we have one less thing to worry about.

However, not all data can easily be represented as fixed-length vectors. While we might expect microscope images to come from standard equipment, we cannot expect images mined from the Internet to all show up with the same resolution or shape. For images, we might consider cropping them all to a standard size, but that strategy only gets us so far. We risk losing information in the cropped out portions. Moreover, text data resists fixed-length representations even more stubbornly. Consider the customer reviews left on e-commerce sites like Amazon, IMDB, or TripAdvisor. Some are short: "it stinks!". Others ramble for pages. One major advantage of deep learning over traditional methods is the comparative grace with which modern models can handle *varying-length* data.

Generally, the more data we have, the easier our job becomes. When we have more data, we can train more powerful models and rely less heavily on pre-conceived assumptions. The regime change from (comparatively) small to big data is a major contributor to the success of modern deep learning. To drive the point home, many of the most exciting models in deep learning do not work without large datasets. Some others work in the low-data regime, but are no better than traditional approaches.

Finally, it is not enough to have lots of data and to process it cleverly. We need the *right* data. If the data is full of mistakes, or if the chosen features are not predictive of the target quantity of interest, learning is going to fail. The situation is captured well by the cliché: *garbage in, garbage out*. Moreover, poor predictive performance is not the only potential consequence. In sensitive applications of machine learning, like predictive policing, resumé screening, and risk models used for lending, we must be especially alert to the consequences of garbage data. One common failure mode occurs in datasets where some groups of people are unrepresented in the training data. Imagine applying a skin cancer recognition system in the wild that had never seen black skin before. Failure can also occur when the data does not merely under-represent some groups but reflects societal prejudices. For example, if past hiring decisions are used to train a predictive model that will be used to screen resumes, then machine learning models could inadvertently

capture and automate historical injustices. Note that this can all happen without the data scientist actively conspiring, or even being aware.

1.2.2 Models

Most machine learning involves *transforming* the data in some sense. We might want to build a system that ingests photos and predicts *smiley-ness*. Alternatively, we might want to ingest a set of sensor readings and predict how *normal* vs *anomalous* the readings are. By *model*, we denote the computational machinery for ingesting data of one type, and spitting out predictions of a possibly different type. In particular, we are interested in statistical models that can be estimated from data. While simple models are perfectly capable of addressing appropriately simple problems the problems that we focus on in this book stretch the limits of classical methods. Deep learning is differentiated from classical approaches principally by the set of powerful models that it focuses on. These models consist of many successive transformations of the data that are chained together top to bottom, thus the name *deep learning*. On our way to discussing deep neural networks, we will discuss some more traditional methods.

1.2.3 Objective functions

Earlier, we introduced machine learning as "learning from experience". By *learning* here, we mean *improving* at some task over time. But who is to say what constitutes an improvement? You might imagine that we could propose to update our model, and some people might disagree on whether the proposed update constituted an improvement or a decline.

In order to develop a formal mathematical system of learning machines, we need to have formal measures of how good (or bad) our models are. In machine learning, and optimization more generally, we call these objective functions. By convention, we usually define objective functions so that *lower* is *better*. This is merely a convention. You can take any function f for which higher is better, and turn it into a new function f' that is qualitatively identical but for which lower is better by setting $f' = -f$. Because lower is better, these functions are sometimes called *loss functions* or *cost functions*.

When trying to predict numerical values, the most common objective function is squared error $(y - \hat{y})^2$. For classification, the most common objective is to minimize error rate, i.e., the fraction of instances on which our predictions disagree with the ground truth. Some objectives (like squared error) are easy to optimize. Others (like error rate) are difficult to optimize directly, owing to non-differentiability or other complications. In these cases, it is common to optimize a *surrogate objective*.

Typically, the loss function is defined with respect to the model's parameters and depends upon the dataset. The best values of our model's parameters are learned by minimizing the loss incurred on a *training set* consisting of some number of *examples* collected for training. However, doing well on the training data does not guarantee that we will do well on (unseen) test data. So we will typically want to split the available data into two partitions: the training data (for fitting model parameters) and the test data (which is held out for evaluation), reporting the following two quantities:

- **Training Error:** The error on that data on which the model was trained. You could think of this as being like a student's scores on practice exams used to prepare for some real exam. Even if the results are encouraging, that does not guarantee success on the final exam.
- **Test Error:** This is the error incurred on an unseen test set. This can deviate significantly from the training error. When a model performs well on the training data but fails to gen-

eralize to unseen data, we say that it is *overfitting*. In real-life terms, this is like flunking the real exam despite doing well on practice exams.

1.2.4 Optimization algorithms

Once we have got some data source and representation, a model, and a well-defined objective function, we need an algorithm capable of searching for the best possible parameters for minimizing the loss function. The most popular optimization algorithms for neural networks follow an approach called gradient descent. In short, at each step, they check to see, for each parameter, which way the training set loss would move if you perturbed that parameter just a small amount. They then update the parameter in the direction that reduces the loss.

1.3 Kinds of Machine Learning

In the following sections, we discuss a few *kinds* of machine learning problems in greater detail. We begin with a list of *objectives*, i.e., a list of things that we would like machine learning to do. Note that the objectives are complemented with a set of techniques of *how* to accomplish them, including types of data, models, training techniques, etc. The list below is just a sampling of the problems ML can tackle to motivate the reader and provide us with some common language for when we talk about more problems throughout the book.

1.3.1 Supervised learning

Supervised learning addresses the task of predicting *targets* given *inputs*. The targets, which we often call *labels*, are generally denoted by y. The input data, also called the *features* or covariates, are typically denoted $\mathbf{x}$. Each (input, target) pair is called an *example* or *instance*. Sometimes, when the context is clear, we may use the term examples, to refer to a collection of inputs, even when the corresponding targets are unknown. We denote any particular instance with a subscript, typically i, for instance $(\mathbf{x}_i, y_i)$. A dataset is a collection of n instances $\{\mathbf{x}_i, y_i\}_{i=1}^n$. Our goal is to produce a model f_θ that maps any input $\mathbf{x}_i$ to a prediction $f_\theta(\mathbf{x}_i)$.

To ground this description in a concrete example, if we were working in healthcare, then we might want to predict whether or not a patient would have a heart attack. This observation, *heart attack* or *no heart attack*, would be our label y. The input data $\mathbf{x}$ might be vital signs such as heart rate, diastolic and systolic blood pressure, etc.

The supervision comes into play because for choosing the parameters θ, we (the supervisors) provide the model with a dataset consisting of *labeled examples* $(\mathbf{x}_i, y_i)$, where each example $\mathbf{x}_i$ is matched with the correct label.

In probabilistic terms, we typically are interested in estimating the conditional probability $P(y|x)$. While it is just one among several paradigms within machine learning, supervised learning accounts for the majority of successful applications of machine learning in industry. Partly, that is because many important tasks can be described crisply as estimating the probability of something unknown given a particular set of available data:

- Predict cancer vs not cancer, given a CT image.
- Predict the correct translation in French, given a sentence in English.
- Predict the price of a stock next month based on this month's financial reporting data.

Even with the simple description "predict targets from inputs" supervised learning can take a great many forms and require a great many modeling decisions, depending on (among other considerations) the type, size, and the number of inputs and outputs. For example, we use different models to process sequences (like strings of text or time series data) and for processing fixed-length vector representations. We will visit many of these problems in depth throughout the first 9 parts of this book.

Informally, the learning process looks something like this: Grab a big collection of examples for which the covariates are known and select from them a random subset, acquiring the ground truth labels for each. Sometimes these labels might be available data that has already been collected (e.g., did a patient die within the following year?) and other times we might need to employ human annotators to label the data, (e.g., assigning images to categories).

Together, these inputs and corresponding labels comprise the training set. We feed the training dataset into a supervised learning algorithm, a function that takes as input a dataset and outputs another function, *the learned model*. Finally, we can feed previously unseen inputs to the learned model, using its outputs as predictions of the corresponding label. The full process is drawn in Fig. 1.3.1.

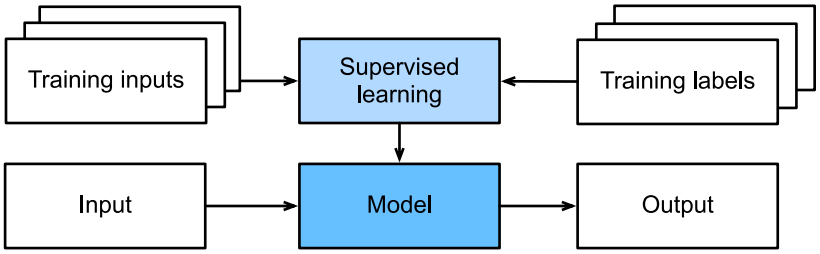


Fig. 1.3.1: Supervised learning.

Regression

Perhaps the simplest supervised learning task to wrap your head around is *regression*. Consider, for example, a set of data harvested from a database of home sales. We might construct a table, where each row corresponds to a different house, and each column corresponds to some relevant attribute, such as the square footage of a house, the number of bedrooms, the number of bathrooms, and the number of minutes (walking) to the center of town. In this dataset, each *example* would be a specific house, and the corresponding *feature vector* would be one row in the table.

If you live in New York or San Francisco, and you are not the CEO of Amazon, Google, Microsoft, or Facebook, the (sq. footage, no. of bedrooms, no. of bathrooms, walking distance) feature vector for your home might look something like: $[100, 0, .5, 60]$. However, if you live in Pittsburgh, it might look more like $[3000, 4, 3, 10]$. Feature vectors like this are essential for most classic machine learning algorithms. We will continue to denote the feature vector corresponding to any example i as $\mathbf{x}_i$ and we can compactly refer to the full table containing all of the feature vectors as X.

What makes a problem a *regression* is actually the outputs. Say that you are in the market for a new home. You might want to estimate the fair market value of a house, given some features like these. The target value, the price of sale, is a *real number*. If you remember the formal definition of the reals you might be scratching your head now. Homes probably never sell for fractions of a cent, let alone prices expressed as irrational numbers. In cases like this, when the target is actually discrete, but where the rounding takes place on a sufficiently fine scale, we will abuse language just a bit and continue to describe our outputs and targets as real-valued numbers.

We denote any individual target y_i (corresponding to example $\mathbf{x}_i$) and the set of all targets $\mathbf{y}$ (corresponding to all examples X). When our targets take on arbitrary values in some range, we call this a regression problem. Our goal is to produce a model whose predictions closely approximate the actual target values. We denote the predicted target for any instance $\hat{y}_i$. Do not worry if the notation is bogging you down. We will unpack it more thoroughly in the subsequent chapters.

Lots of practical problems are well-described regression problems. Predicting the rating that a user will assign to a movie can be thought of as a regression problem and if you designed a great algorithm to accomplish this feat in 2009, you might have won the 1-million-dollar Netflix prize[13]. Predicting the length of stay for patients in the hospital is also a regression problem. A good rule of thumb is that any *How much?* or *How many?* problem should suggest regression.

- "How many hours will this surgery take?": *regression*
- "How many dogs are in this photo?": *regression*.

However, if you can easily pose your problem as "Is this a _ ?", then it is likely, classification, a different kind of supervised problem that we will cover next. Even if you have never worked with machine learning before, you have probably worked through a regression problem informally. Imagine, for example, that you had your drains repaired and that your contractor spent $x_1 = 3$ hours removing gunk from your sewage pipes. Then he sent you a bill of $y_1 = \$350$. Now imagine that your friend hired the same contractor for $x_2 = 2$ hours and that he received a bill of $y_2 = \$250$. If someone then asked you how much to expect on their upcoming gunk-removal invoice you might make some reasonable assumptions, such as more hours worked costs more dollars. You might also assume that there is some base charge and that the contractor then charges per hour. If these assumptions held true, then given these two data examples, you could already identify the contractor's pricing structure: \$100 per hour plus \$50 to show up at your house. If you followed that much then you already understand the high-level idea behind linear regression (and you just implicitly designed a linear model with a bias term).

In this case, we could produce the parameters that exactly matched the contractor's prices. Sometimes that is not possible, e.g., if some of the variance owes to some factors besides your two features. In these cases, we will try to learn models that minimize the distance between our predictions and the observed values. In most of our chapters, we will focus on one of two very common losses, the L1 loss where

$$l(y, y') = \sum_i |y_i - y_i'| \tag{1.3.1}$$

and the least mean squares loss, or L_2 loss where

$$l(y, y') = \sum_i (y_i - y_i')^2. \tag{1.3.2}$$

As we will see later, the L_2 loss corresponds to the assumption that our data was corrupted by Gaussian noise, whereas the L_1 loss corresponds to an assumption of noise from a Laplace distribution.

[13] https://en.wikipedia.org/wiki/Netflix_Prize

Classification

While regression models are great for addressing *how many?* questions, lots of problems do not bend comfortably to this template. For example, a bank wants to add check scanning to its mobile app. This would involve the customer snapping a photo of a check with their smart phone's camera and the machine learning model would need to be able to automatically understand text seen in the image. It would also need to understand hand-written text to be even more robust. This kind of system is referred to as optical character recognition (OCR), and the kind of problem it addresses is called *classification*. It is treated with a different set of algorithms than those used for regression (although many techniques will carry over).

In classification, we want our model to look at a feature vector, e.g., the pixel values in an image, and then predict which category (formally called *classes*), among some (discrete) set of options, an example belongs. For hand-written digits, we might have 10 classes, corresponding to the digits 0 through 9. The simplest form of classification is when there are only two classes, a problem which we call binary classification. For example, our dataset X could consist of images of animals and our *labels* Y might be the classes $\{\text{cat}, \text{dog}\}$. While in regression, we sought a *regressor* to output a real value $\hat{y}$, in classification, we seek a *classifier*, whose output $\hat{y}$ is the predicted class assignment.

For reasons that we will get into as the book gets more technical, it can be hard to optimize a model that can only output a hard categorical assignment, e.g., either *cat* or *dog*. In these cases, it is usually much easier to instead express our model in the language of probabilities. Given an example x, our model assigns a probability $\hat{y}_k$ to each label k. Because these are probabilities, they need to be positive numbers and add up to 1 and thus we only need $K - 1$ numbers to assign probabilities of K categories. This is easy to see for binary classification. If there is a 0.6 (60%) probability that an unfair coin comes up heads, then there is a 0.4 (40%) probability that it comes up tails. Returning to our animal classification example, a classifier might see an image and output the probability that the image is a cat $P(y = \text{cat} \mid x) = 0.9$. We can interpret this number by saying that the classifier is 90% sure that the image depicts a cat. The magnitude of the probability for the predicted class conveys one notion of uncertainty. It is not the only notion of uncertainty and we will discuss others in more advanced chapters.

When we have more than two possible classes, we call the problem *multiclass classification*. Common examples include hand-written character recognition `[0, 1, 2, 3 ... 9, a, b, c, ...]`. While we attacked regression problems by trying to minimize the L_1 or L_2 loss functions, the common loss function for classification problems is called cross-entropy.

Note that the most likely class is not necessarily the one that you are going to use for your decision. Assume that you find this beautiful mushroom in your backyard as shown in Fig. 1.3.2.

Fig. 1.3.2: Death cap—do not eat!

Now, assume that you built a classifier and trained it to predict if a mushroom is poisonous based on a photograph. Say our poison-detection classifier outputs $P(y = \text{deathcap}|\text{image}) = 0.2$. In other words, the classifier is 80% sure that our mushroom *is not* a death cap. Still, you would have to be a fool to eat it. That is because the certain benefit of a delicious dinner is not worth a 20% risk of dying from it. In other words, the effect of the *uncertain risk* outweighs the benefit by far. We can look at this more formally. Basically, we need to compute the expected risk that we incur, i.e., we need to multiply the probability of the outcome with the benefit (or harm) associated with it:

$$L(\text{action}|x) = E_{y \sim p(y|x)}[\text{loss}(\text{action}, y)]. \tag{1.3.3}$$

Hence, the loss L incurred by eating the mushroom is $L(a = \text{eat}|x) = 0.2 * \infty + 0.8 * 0 = \infty$, whereas the cost of discarding it is $L(a = \text{discard}|x) = 0.2 * 0 + 0.8 * 1 = 0.8$.

Our caution was justified: as any mycologist would tell us, the above mushroom actually *is* a death cap. Classification can get much more complicated than just binary, multiclass, or even multi-label classification. For instance, there are some variants of classification for addressing hierarchies. Hierarchies assume that there exist some relationships among the many classes. So not all errors are equal—if we must err, we would prefer to misclassify to a related class rather than to a distant class. Usually, this is referred to as *hierarchical classification*. One early example is due to Linnaeus[14], who organized the animals in a hierarchy.

In the case of animal classification, it might not be so bad to mistake a poodle for a schnauzer, but our model would pay a huge penalty if it confused a poodle for a dinosaur. Which hierarchy is relevant might depend on how you plan to use the model. For example, rattle snakes and garter snakes might be close on the phylogenetic tree, but mistaking a rattler for a garter could be deadly.

[14] https://en.wikipedia.org/wiki/Carl_Linnaeus

Tagging

Some classification problems do not fit neatly into the binary or multiclass classification setups. For example, we could train a normal binary classifier to distinguish cats from dogs. Given the current state of computer vision, we can do this easily, with off-the-shelf tools. Nonetheless, no matter how accurate our model gets, we might find ourselves in trouble when the classifier encounters an image of the Town Musicians of Bremen.

Fig. 1.3.3: A cat, a rooster, a dog and a donkey

As you can see, there is a cat in the picture, and a rooster, a dog, a donkey, and a bird, with some trees in the background. Depending on what we want to do with our model ultimately, treating this as a binary classification problem might not make a lot of sense. Instead, we might want to give the model the option of saying the image depicts a cat *and* a dog *and* a donkey *and* a rooster *and* a bird.

The problem of learning to predict classes that are *not mutually exclusive* is called multi-label classification. Auto-tagging problems are typically best described as multi-label classification problems. Think of the tags people might apply to posts on a tech blog, e.g., "machine learning", "technology", "gadgets", "programming languages", "linux", "cloud computing", "AWS". A typical article might have 5-10 tags applied because these concepts are correlated. Posts about "cloud computing" are likely to mention "AWS" and posts about "machine learning" could also deal with "programming languages".

We also have to deal with this kind of problem when dealing with the biomedical literature, where correctly tagging articles is important because it allows researchers to do exhaustive reviews of the literature. At the National Library of Medicine, a number of professional annotators go over

each article that gets indexed in PubMed to associate it with the relevant terms from MeSH, a collection of roughly 28k tags. This is a time-consuming process and the annotators typically have a one year lag between archiving and tagging. Machine learning can be used here to provide provisional tags until each article can have a proper manual review. Indeed, for several years, the BioASQ organization has hosted a competition[15] to do precisely this.

Search and ranking

Sometimes we do not just want to assign each example to a bucket or to a real value. In the field of information retrieval, we want to impose a ranking on a set of items. Take web search for example, the goal is less to determine whether a particular page is relevant for a query, but rather, which one of the plethora of search results is *most relevant* for a particular user. We really care about the ordering of the relevant search results and our learning algorithm needs to produce ordered subsets of elements from a larger set. In other words, if we are asked to produce the first 5 letters from the alphabet, there is a difference between returning `A B C D E` and `C A B E D`. Even if the result set is the same, the ordering within the set matters.

One possible solution to this problem is to first assign to every element in the set a corresponding relevance score and then to retrieve the top-rated elements. PageRank[16], the original secret sauce behind the Google search engine was an early example of such a scoring system but it was peculiar in that it did not depend on the actual query. Here they relied on a simple relevance filter to identify the set of relevant items and then on PageRank to order those results that contained the query term. Nowadays, search engines use machine learning and behavioral models to obtain query-dependent relevance scores. There are entire academic conferences devoted to this subject.

Recommender systems

Recommender systems are another problem setting that is related to search and ranking. The problems are similar insofar as the goal is to display a set of relevant items to the user. The main difference is the emphasis on *personalization* to specific users in the context of recommender systems. For instance, for movie recommendations, the results page for a SciFi fan and the results page for a connoisseur of Peter Sellers comedies might differ significantly. Similar problems pop up in other recommendation settings, e.g., for retail products, music, or news recommendation.

In some cases, customers provide explicit feedback communicating how much they liked a particular product (e.g., the product ratings and reviews on Amazon, IMDB, GoodReads, etc.). In some other cases, they provide implicit feedback, e.g., by skipping titles on a playlist, which might indicate dissatisfaction but might just indicate that the song was inappropriate in context. In the simplest formulations, these systems are trained to estimate some score y_{ij}, such as an estimated rating or the probability of purchase, given a user u_i and product p_j.

Given such a model, then for any given user, we could retrieve the set of objects with the largest scores y_{ij}, which could then be recommended to the customer. Production systems are considerably more advanced and take detailed user activity and item characteristics into account when computing such scores. Fig. 1.3.4 is an example of deep learning books recommended by Amazon based on personalization algorithms tuned to capture the author's preferences.

[15] http://bioasq.org/
[16] https://en.wikipedia.org/wiki/PageRank

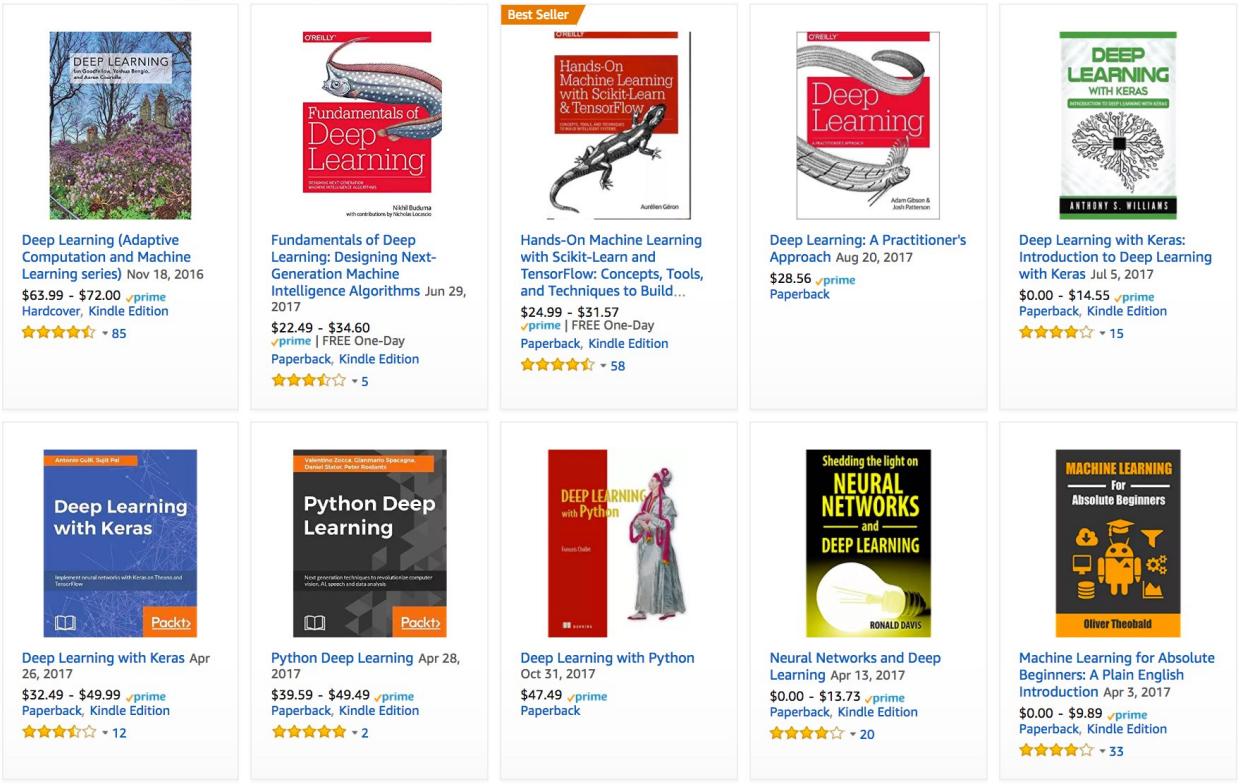


Fig. 1.3.4: Deep learning books recommended by Amazon.

Despite their tremendous economic value, recommendation systems naively built on top of predictive models suffer some serious conceptual flaws. To start, we only observe *censored feedback*. Users preferentially rate movies that they feel strongly about: you might notice that items receive many 5 and 1 star ratings but that there are conspicuously few 3-star ratings. Moreover, current purchase habits are often a result of the recommendation algorithm currently in place, but learning algorithms do not always take this detail into account. Thus it is possible for feedback loops to form where a recommender system preferentially pushes an item that is then taken to be better (due to greater purchases) and in turn is recommended even more frequently. Many of these problems about how to deal with censoring, incentives, and feedback loops, are important open research questions.

Sequence Learning

So far, we have looked at problems where we have some fixed number of inputs and produce a fixed number of outputs. Before we considered predicting home prices from a fixed set of features: square footage, number of bedrooms, number of bathrooms, walking time to downtown. We also discussed mapping from an image (of fixed dimension) to the predicted probabilities that it belongs to each of a fixed number of classes, or taking a user ID and a product ID, and predicting a star rating. In these cases, once we feed our fixed-length input into the model to generate an output, the model immediately forgets what it just saw.

This might be fine if our inputs truly all have the same dimensions and if successive inputs truly have nothing to do with each other. But how would we deal with video snippets? In this case, each snippet might consist of a different number of frames. And our guess of what is going on in each frame might be much stronger if we take into account the previous or succeeding frames.

Same goes for language. One popular deep learning problem is machine translation: the task of ingesting sentences in some source language and predicting their translation in another language.

These problems also occur in medicine. We might want a model to monitor patients in the intensive care unit and to fire off alerts if their risk of death in the next 24 hours exceeds some threshold. We definitely would not want this model to throw away everything it knows about the patient history each hour and just make its predictions based on the most recent measurements.

These problems are among the most exciting applications of machine learning and they are instances of *sequence learning*. They require a model to either ingest sequences of inputs or to emit sequences of outputs (or both!). These latter problems are sometimes referred to as `seq2seq` problems. Language translation is a `seq2seq` problem. Transcribing text from the spoken speech is also a `seq2seq` problem. While it is impossible to consider all types of sequence transformations, a number of special cases are worth mentioning:

Tagging and Parsing. This involves annotating a text sequence with attributes. In other words, the number of inputs and outputs is essentially the same. For instance, we might want to know where the verbs and subjects are. Alternatively, we might want to know which words are the named entities. In general, the goal is to decompose and annotate text based on structural and grammatical assumptions to get some annotation. This sounds more complex than it actually is. Below is a very simple example of annotating a sentence with tags indicating which words refer to named entities.

```
Tom has dinner in Washington with Sally.
Ent  -    -    -     Ent      -    Ent
```

Automatic Speech Recognition. With speech recognition, the input sequence x is an audio recording of a speaker (shown in Fig. 1.3.5), and the output y is the textual transcript of what the speaker said. The challenge is that there are many more audio frames (sound is typically sampled at 8kHz or 16kHz) than text, i.e., there is no 1:1 correspondence between audio and text, since thousands of samples correspond to a single spoken word. These are `seq2seq` problems where the output is much shorter than the input.

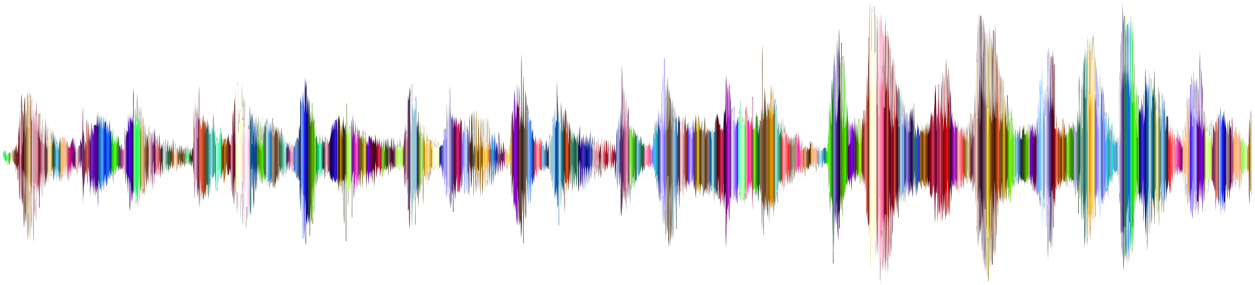


Fig. 1.3.5: `-D-e-e-p- L-ea-r-ni-ng-`

Text to Speech. Text-to-Speech (TTS) is the inverse of speech recognition. In other words, the input x is text and the output y is an audio file. In this case, the output is *much longer* than the input. While it is easy for *humans* to recognize a bad audio file, this is not quite so trivial for computers.

Machine Translation. Unlike the case of speech recognition, where corresponding inputs and outputs occur in the same order (after alignment), in machine translation, order inversion can be vital. In other words, while we are still converting one sequence into another, neither the number of inputs and outputs nor the order of corresponding data examples are assumed to be the same.

Consider the following illustrative example of the peculiar tendency of Germans to place the verbs at the end of sentences.

```
German:           Haben Sie sich schon dieses grossartige Lehrwerk angeschaut?
English:          Did you already check out this excellent tutorial?
Wrong alignment:  Did you yourself already this excellent tutorial looked-at?
```

Many related problems pop up in other learning tasks. For instance, determining the order in which a user reads a Webpage is a two-dimensional layout analysis problem. Dialogue problems exhibit all kinds of additional complications, where determining what to say next requires taking into account real-world knowledge and the prior state of the conversation across long temporal distances. This is an active area of research.

1.3.2 Unsupervised learning

All the examples so far were related to *Supervised Learning*, i.e., situations where we feed the model a giant dataset containing both the features and corresponding target values. You could think of the supervised learner as having an extremely specialized job and an extremely banal boss. The boss stands over your shoulder and tells you exactly what to do in every situation until you learn to map from situations to actions. Working for such a boss sounds pretty lame. On the other hand, it is easy to please this boss. You just recognize the pattern as quickly as possible and imitate their actions.

In a completely opposite way, it could be frustrating to work for a boss who has no idea what they want you to do. However, if you plan to be a data scientist, you'd better get used to it. The boss might just hand you a giant dump of data and tell you to *do some data science with it!* This sounds vague because it is. We call this class of problems *unsupervised learning*, and the type and number of questions we could ask is limited only by our creativity. We will address a number of unsupervised learning techniques in later chapters. To whet your appetite for now, we describe a few of the questions you might ask:

- Can we find a small number of prototypes that accurately summarize the data? Given a set of photos, can we group them into landscape photos, pictures of dogs, babies, cats, mountain peaks, etc.? Likewise, given a collection of users' browsing activity, can we group them into users with similar behavior? This problem is typically known as *clustering*.
- Can we find a small number of parameters that accurately capture the relevant properties of the data? The trajectories of a ball are quite well described by velocity, diameter, and mass of the ball. Tailors have developed a small number of parameters that describe human body shape fairly accurately for the purpose of fitting clothes. These problems are referred to as *subspace estimation* problems. If the dependence is linear, it is called *principal component analysis*.
- Is there a representation of (arbitrarily structured) objects in Euclidean space (i.e., the space of vectors in $\mathbb{R}^n$) such that symbolic properties can be well matched? This is called *representation learning* and it is used to describe entities and their relations, such as Rome $-$ Italy $+$ France $=$ Paris.
- Is there a description of the root causes of much of the data that we observe? For instance, if we have demographic data about house prices, pollution, crime, location, education, salaries, etc., can we discover how they are related simply based on empirical data? The fields concerned with *causality* and *probabilistic graphical models* address this problem.

- Another important and exciting recent development in unsupervised learning is the advent of *generative adversarial networks* (GANs). These give us a procedural way to synthesize data, even complicated structured data like images and audio. The underlying statistical mechanisms are tests to check whether real and fake data are the same. We will devote a few notebooks to them.

1.3.3 Interacting with an Environment

So far, we have not discussed where data actually comes from, or what actually *happens* when a machine learning model generates an output. That is because supervised learning and unsupervised learning do not address these issues in a very sophisticated way. In either case, we grab a big pile of data upfront, then set our pattern recognition machines in motion without ever interacting with the environment again. Because all of the learning takes place after the algorithm is disconnected from the environment, this is sometimes called *offline learning*. For supervised learning, the process looks like Fig. 1.3.6.

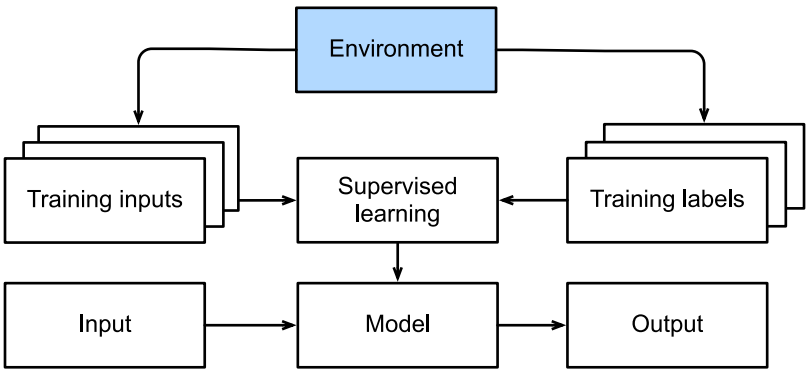


Fig. 1.3.6: Collect data for supervised learning from an environment.

This simplicity of offline learning has its charms. The upside is we can worry about pattern recognition in isolation, without any distraction from these other problems. But the downside is that the problem formulation is quite limiting. If you are more ambitious, or if you grew up reading Asimov's Robot Series, then you might imagine artificially intelligent bots capable not only of making predictions, but of taking actions in the world. We want to think about intelligent *agents*, not just predictive *models*. That means we need to think about choosing *actions*, not just making *predictions*. Moreover, unlike predictions, actions actually impact the environment. If we want to train an intelligent agent, we must account for the way its actions might impact the future observations of the agent.

Considering the interaction with an environment opens a whole set of new modeling questions. Does the environment:

- Remember what we did previously?
- Want to help us, e.g., a user reading text into a speech recognizer?
- Want to beat us, i.e., an adversarial setting like spam filtering (against spammers) or playing a game (vs an opponent)?
- Not care (as in many cases)?
- Have shifting dynamics (does future data always resemble the past or do the patterns change over time, either naturally or in response to our automated tools)?

This last question raises the problem of *distribution shift*, (when training and test data are different). It is a problem that most of us have experienced when taking exams written by a lecturer, while the homeworks were composed by his TAs. We will briefly describe reinforcement learning and adversarial learning, two settings that explicitly consider interaction with an environment.

1.3.4 Reinforcement learning

If you are interested in using machine learning to develop an agent that interacts with an environment and takes actions, then you are probably going to wind up focusing on *reinforcement learning* (RL). This might include applications to robotics, to dialogue systems, and even to developing AI for video games. *Deep reinforcement learning* (DRL), which applies deep neural networks to RL problems, has surged in popularity. The breakthrough deep Q-network that beat humans at Atari games using only the visual input[17], and the AlphaGo program that dethroned the world champion at the board game Go[18] are two prominent examples.

Reinforcement learning gives a very general statement of a problem, in which an agent interacts with an environment over a series of *time steps*. At each time step t, the agent receives some observation o_t from the environment and must choose an action a_t that is subsequently transmitted back to the environment via some mechanism (sometimes called an actuator). Finally, the agent receives a reward r_t from the environment. The agent then receives a subsequent observation, and chooses a subsequent action, and so on. The behavior of an RL agent is governed by a *policy*. In short, a *policy* is just a function that maps from observations (of the environment) to actions. The goal of reinforcement learning is to produce a good policy.

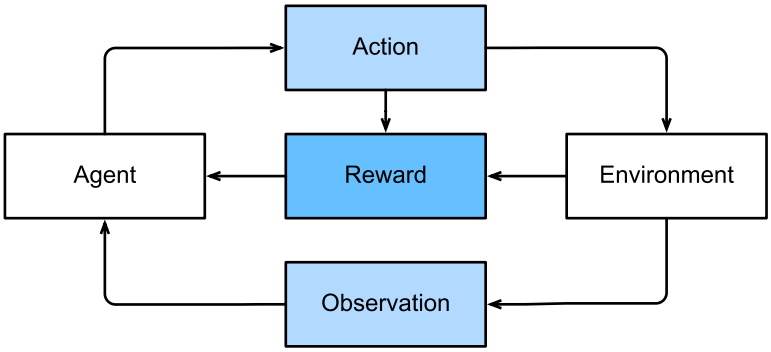


Fig. 1.3.7: The interaction between reinforcement learning and an environment.

It is hard to overstate the generality of the RL framework. For example, we can cast any supervised learning problem as an RL problem. Say we had a classification problem. We could create an RL agent with one *action* corresponding to each class. We could then create an environment which gave a reward that was exactly equal to the loss function from the original supervised problem.

That being said, RL can also address many problems that supervised learning cannot. For example, in supervised learning we always expect that the training input comes associated with the correct label. But in RL, we do not assume that for each observation, the environment tells us the optimal action. In general, we just get some reward. Moreover, the environment may not even tell us which actions led to the reward.

Consider for example the game of chess. The only real reward signal comes at the end of the game when we either win, which we might assign a reward of 1, or when we lose, which we could

[17] https://www.wired.com/2015/02/google-ai-plays-atari-like-pros/
[18] https://www.wired.com/2017/05/googles-alphago-trounces-humans-also-gives-boost/

assign a reward of -1. So reinforcement learners must deal with the *credit assignment problem*: determining which actions to credit or blame for an outcome. The same goes for an employee who gets a promotion on October 11. That promotion likely reflects a large number of well-chosen actions over the previous year. Getting more promotions in the future requires figuring out what actions along the way led to the promotion.

Reinforcement learners may also have to deal with the problem of partial observability. That is, the current observation might not tell you everything about your current state. Say a cleaning robot found itself trapped in one of many identical closets in a house. Inferring the precise location (and thus state) of the robot might require considering its previous observations before entering the closet.

Finally, at any given point, reinforcement learners might know of one good policy, but there might be many other better policies that the agent has never tried. The reinforcement learner must constantly choose whether to *exploit* the best currently-known strategy as a policy, or to *explore* the space of strategies, potentially giving up some short-run reward in exchange for knowledge.

MDPs, bandits, and friends

The general reinforcement learning problem is a very general setting. Actions affect subsequent observations. Rewards are only observed corresponding to the chosen actions. The environment may be either fully or partially observed. Accounting for all this complexity at once may ask too much of researchers. Moreover, not every practical problem exhibits all this complexity. As a result, researchers have studied a number of *special cases* of reinforcement learning problems.

When the environment is fully observed, we call the RL problem a *Markov Decision Process* (MDP). When the state does not depend on the previous actions, we call the problem a *contextual bandit problem*. When there is no state, just a set of available actions with initially unknown rewards, this problem is the classic *multi-armed bandit problem*.

1.4 Roots

Although many deep learning methods are recent inventions, humans have held the desire to analyze data and to predict future outcomes for centuries. In fact, much of natural science has its roots in this. For instance, the Bernoulli distribution is named after Jacob Bernoulli (1655-1705)[19], and the Gaussian distribution was discovered by Carl Friedrich Gauss (1777-1855)[20]. He invented, for instance, the least mean squares algorithm, which is still used today for countless problems from insurance calculations to medical diagnostics. These tools gave rise to an experimental approach in the natural sciences—for instance, Ohm's law relating current and voltage in a resistor is perfectly described by a linear model.

Even in the middle ages, mathematicians had a keen intuition of estimates. For instance, the geometry book of Jacob Köbel (1460-1533)[21] illustrates averaging the length of 16 adult men's feet to obtain the average foot length.

[19] https://en.wikipedia.org/wiki/Jacob_Bernoulli
[20] https://en.wikipedia.org/wiki/Carl_Friedrich_Gauss
[21] https://www.maa.org/press/periodicals/convergence/mathematical-treasures-jacob-kobels-geometry

Fig. 1.4.1: Estimating the length of a foot

Fig. 1.4.1 illustrates how this estimator works. The 16 adult men were asked to line up in a row, when leaving church. Their aggregate length was then divided by 16 to obtain an estimate for what now amounts to 1 foot. This "algorithm" was later improved to deal with misshapen feet—the 2 men with the shortest and longest feet respectively were sent away, averaging only over the remainder. This is one of the earliest examples of the trimmed mean estimate.

Statistics really took off with the collection and availability of data. One of its titans, Ronald Fisher (1890-1962)[22], contributed significantly to its theory and also its applications in genetics. Many of his algorithms (such as Linear Discriminant Analysis) and formula (such as the Fisher Information Matrix) are still in frequent use today (even the Iris dataset that he released in 1936 is still used sometimes to illustrate machine learning algorithms). Fisher was also a proponent of eugenics, which should remind us that the morally dubious use of data science has as long and enduring a history as its productive use in industry and the natural sciences.

A second influence for machine learning came from Information Theory (Claude Shannon, 1916-2001)[23] and the Theory of computation via Alan Turing (1912-1954)[24]. Turing posed the question "can machines think?" in his famous paper Computing machinery and intelligence[25] (Mind, October 1950). In what he described as the Turing test, a machine can be considered intelligent if it is difficult for a human evaluator to distinguish between the replies from a machine and a human based on textual interactions.

Another influence can be found in neuroscience and psychology. After all, humans clearly exhibit intelligent behavior. It is thus only reasonable to ask whether one could explain and possibly re-

[22] https://en.wikipedia.org/wiki/Ronald_Fisher
[23] https://en.wikipedia.org/wiki/Claude_Shannon
[24] https://en.wikipedia.org/wiki/Alan_Turing
[25] https://en.wikipedia.org/wiki/Computing_Machinery_and_Intelligence

verse engineer this capacity. One of the oldest algorithms inspired in this fashion was formulated by Donald Hebb (1904-1985)[26]. In his groundbreaking book The Organization of Behavior (Hebb & Hebb, 1949), he posited that neurons learn by positive reinforcement. This became known as the Hebbian learning rule. It is the prototype of Rosenblatt's perceptron learning algorithm and it laid the foundations of many stochastic gradient descent algorithms that underpin deep learning today: reinforce desirable behavior and diminish undesirable behavior to obtain good settings of the parameters in a neural network.

Biological inspiration is what gave *neural networks* their name. For over a century (dating back to the models of Alexander Bain, 1873 and James Sherrington, 1890), researchers have tried to assemble computational circuits that resemble networks of interacting neurons. Over time, the interpretation of biology has become less literal but the name stuck. At its heart, lie a few key principles that can be found in most networks today:

- The alternation of linear and nonlinear processing units, often referred to as *layers*.
- The use of the chain rule (also known as *backpropagation*) for adjusting parameters in the entire network at once.

After initial rapid progress, research in neural networks languished from around 1995 until 2005. This was due to a number of reasons. Training a network is computationally very expensive. While RAM was plentiful at the end of the past century, computational power was scarce. Second, datasets were relatively small. In fact, Fisher's Iris dataset from 1932 was a popular tool for testing the efficacy of algorithms. MNIST with its 60,000 handwritten digits was considered huge.

Given the scarcity of data and computation, strong statistical tools such as Kernel Methods, Decision Trees and Graphical Models proved empirically superior. Unlike neural networks, they did not require weeks to train and provided predictable results with strong theoretical guarantees.

1.5 The Road to Deep Learning

Much of this changed with the ready availability of large amounts of data, due to the World Wide Web, the advent of companies serving hundreds of millions of users online, a dissemination of cheap, high-quality sensors, cheap data storage (Kryder's law), and cheap computation (Moore's law), in particular in the form of GPUs, originally engineered for computer gaming. Suddenly algorithms and models that seemed computationally infeasible became relevant (and vice versa). This is best illustrated in Table 1.5.1.

Table 1.5.1: Dataset vs. computer memory and computational power

Decade	Dataset	Memory	Floating Point Calculations per Second
1970	100 (Iris)	1 KB	100 KF (Intel 8080)
1980	1 K (House prices in Boston)	100 KB	1 MF (Intel 80186)
1990	10 K (optical character recognition)	10 MB	10 MF (Intel 80486)
2000	10 M (web pages)	100 MB	1 GF (Intel Core)
2010	10 G (advertising)	1 GB	1 TF (Nvidia C2050)
2020	1 T (social network)	100 GB	1 PF (Nvidia DGX-2)

It is evident that RAM has not kept pace with the growth in data. At the same time, the increase

[26] https://en.wikipedia.org/wiki/Donald_O._Hebb

in computational power has outpaced that of the data available. This means that statistical models needed to become more memory efficient (this is typically achieved by adding nonlinearities) while simultaneously being able to spend more time on optimizing these parameters, due to an increased compute budget. Consequently, the sweet spot in machine learning and statistics moved from (generalized) linear models and kernel methods to deep networks. This is also one of the reasons why many of the mainstays of deep learning, such as multilayer perceptrons (McCulloch & Pitts, 1943), convolutional neural networks (LeCun et al., 1998), Long Short-Term Memory (Hochreiter & Schmidhuber, 1997), and Q-Learning (Watkins & Dayan, 1992), were essentially "rediscovered" in the past decade, after laying comparatively dormant for considerable time.

The recent progress in statistical models, applications, and algorithms, has sometimes been likened to the Cambrian Explosion: a moment of rapid progress in the evolution of species. Indeed, the state of the art is not just a mere consequence of available resources, applied to decades old algorithms. Note that the list below barely scratches the surface of the ideas that have helped researchers achieve tremendous progress over the past decade.

- Novel methods for capacity control, such as Dropout (Srivastava et al., 2014) have helped to mitigate the danger of overfitting. This was achieved by applying noise injection (Bishop, 1995) throughout the network, replacing weights by random variables for training purposes.
- Attention mechanisms solved a second problem that had plagued statistics for over a century: how to increase the memory and complexity of a system without increasing the number of learnable parameters. (Bahdanau et al., 2014) found an elegant solution by using what can only be viewed as a learnable pointer structure. Rather than having to remember an entire sentence, e.g., for machine translation in a fixed-dimensional representation, all that needed to be stored was a pointer to the intermediate state of the translation process. This allowed for significantly increased accuracy for long sentences, since the model no longer needed to remember the entire sentence before commencing the generation of a new sentence.
- Multi-stage designs, e.g., via the Memory Networks (MemNets) (Sukhbaatar et al., 2015) and the Neural Programmer-Interpreter (Reed & DeFreitas, 2015) allowed statistical modelers to describe iterative approaches to reasoning. These tools allow for an internal state of the deep network to be modified repeatedly, thus carrying out subsequent steps in a chain of reasoning, similar to how a processor can modify memory for a computation.
- Another key development was the invention of GANs (Goodfellow et al., 2014). Traditionally, statistical methods for density estimation and generative models focused on finding proper probability distributions and (often approximate) algorithms for sampling from them. As a result, these algorithms were largely limited by the lack of flexibility inherent in the statistical models. The crucial innovation in GANs was to replace the sampler by an arbitrary algorithm with differentiable parameters. These are then adjusted in such a way that the discriminator (effectively a two-sample test) cannot distinguish fake from real data. Through the ability to use arbitrary algorithms to generate data, it opened up density estimation to a wide variety of techniques. Examples of galloping Zebras (Zhu et al., 2017) and of fake celebrity faces (Karras et al., 2017) are both testimony to this progress. Even amateur doodlers can produce photorealistic images based on just sketches that describe how the layout of a scene looks like (Park et al., 2019).
- In many cases, a single GPU is insufficient to process the large amounts of data available for training. Over the past decade the ability to build parallel distributed training algorithms has improved significantly. One of the key challenges in designing scalable algorithms is that the workhorse of deep learning optimization, stochastic gradient descent, relies on rel-

atively small minibatches of data to be processed. At the same time, small batches limit the efficiency of GPUs. Hence, training on 1024 GPUs with a minibatch size of, say 32 images per batch amounts to an aggregate minibatch of 32k images. Recent work, first by Li (Li, 2017), and subsequently by (You et al., 2017) and (Jia et al., 2018) pushed the size up to 64k observations, reducing training time for ResNet50 on ImageNet to less than 7 minutes. For comparison—initially training times were measured in the order of days.

- The ability to parallelize computation has also contributed quite crucially to progress in reinforcement learning, at least whenever simulation is an option. This has led to significant progress in computers achieving superhuman performance in Go, Atari games, Starcraft, and in physics simulations (e.g., using MuJoCo). See e.g., (Silver et al., 2016) for a description of how to achieve this in AlphaGo. In a nutshell, reinforcement learning works best if plenty of (state, action, reward) triples are available, i.e., whenever it is possible to try out lots of things to learn how they relate to each other. Simulation provides such an avenue.
- Deep Learning frameworks have played a crucial role in disseminating ideas. The first generation of frameworks allowing for easy modeling encompassed Caffe[27], Torch[28], and Theano[29]. Many seminal papers were written using these tools. By now, they have been superseded by TensorFlow[30], often used via its high level API Keras[31], CNTK[32], Caffe 2[33], and Apache MxNet[34]. The third generation of tools, namely imperative tools for deep learning, was arguably spearheaded by Chainer[35], which used a syntax similar to Python NumPy to describe models. This idea was adopted by both PyTorch[36], the Gluon API[37] of MXNet, and Jax[38]. It is the latter group that this course uses to teach deep learning.

The division of labor between systems researchers building better tools and statistical modelers building better networks has greatly simplified things. For instance, training a linear logistic regression model used to be a nontrivial homework problem, worthy to give to new machine learning PhD students at Carnegie Mellon University in 2014. By now, this task can be accomplished with less than 10 lines of code, putting it firmly into the grasp of programmers.

1.6 Success Stories

Artificial Intelligence has a long history of delivering results that would be difficult to accomplish otherwise. For instance, mail is sorted using optical character recognition. These systems have been deployed since the 90s (this is, after all, the source of the famous MNIST and USPS sets of handwritten digits). The same applies to reading checks for bank deposits and scoring creditworthiness of applicants. Financial transactions are checked for fraud automatically. This forms the backbone of many e-commerce payment systems, such as PayPal, Stripe, AliPay, WeChat, Apple, Visa, MasterCard. Computer programs for chess have been competitive for decades. Machine learning feeds search, recommendation, personalization and ranking on the Internet. In other words, artificial intelligence and machine learning are pervasive, albeit often hidden from sight.

[27] https://github.com/BVLC/caffe
[28] https://github.com/torch
[29] https://github.com/Theano/Theano
[30] https://github.com/tensorflow/tensorflow
[31] https://github.com/keras-team/keras
[32] https://github.com/Microsoft/CNTK
[33] https://github.com/caffe2/caffe2
[34] https://github.com/apache/incubator-mxnet
[35] https://github.com/chainer/chainer
[36] https://github.com/pytorch/pytorch
[37] https://github.com/apache/incubator-mxnet
[38] https://github.com/google/jax

It is only recently that AI has been in the limelight, mostly due to solutions to problems that were considered intractable previously.

- Intelligent assistants, such as Apple's Siri, Amazon's Alexa, or Google's assistant are able to answer spoken questions with a reasonable degree of accuracy. This includes menial tasks such as turning on light switches (a boon to the disabled) up to making barber's appointments and offering phone support dialog. This is likely the most noticeable sign that AI is affecting our lives.
- A key ingredient in digital assistants is the ability to recognize speech accurately. Gradually the accuracy of such systems has increased to the point where they reach human parity (Xiong et al., 2018) for certain applications.
- Object recognition likewise has come a long way. Estimating the object in a picture was a fairly challenging task in 2010. On the ImageNet benchmark (Lin et al., 2010) achieved a top-5 error rate of 28%. By 2017, (Hu et al., 2018) reduced this error rate to 2.25%. Similarly, stunning results have been achieved for identifying birds, or diagnosing skin cancer.
- Games used to be a bastion of human intelligence. Starting from TDGammon [23], a program for playing Backgammon using temporal difference (TD) reinforcement learning, algorithmic and computational progress has led to algorithms for a wide range of applications. Unlike Backgammon, chess has a much more complex state space and set of actions. DeepBlue beat Garry Kasparov, Campbell et al. (Campbell et al., 2002), using massive parallelism, special purpose hardware and efficient search through the game tree. Go is more difficult still, due to its huge state space. AlphaGo reached human parity in 2015, (Silver et al., 2016) using Deep Learning combined with Monte Carlo tree sampling. The challenge in Poker was that the state space is large and it is not fully observed (we do not know the opponents' cards). Libratus exceeded human performance in Poker using efficiently structured strategies (Brown & Sandholm, 2017). This illustrates the impressive progress in games and the fact that advanced algorithms played a crucial part in them.
- Another indication of progress in AI is the advent of self-driving cars and trucks. While full autonomy is not quite within reach yet, excellent progress has been made in this direction, with companies such as Tesla, NVIDIA, and Waymo shipping products that enable at least partial autonomy. What makes full autonomy so challenging is that proper driving requires the ability to perceive, to reason and to incorporate rules into a system. At present, deep learning is used primarily in the computer vision aspect of these problems. The rest is heavily tuned by engineers.

Again, the above list barely scratches the surface of where machine learning has impacted practical applications. For instance, robotics, logistics, computational biology, particle physics, and astronomy owe some of their most impressive recent advances at least in parts to machine learning. ML is thus becoming a ubiquitous tool for engineers and scientists.

Frequently, the question of the AI apocalypse, or the AI singularity has been raised in nontechnical articles on AI. The fear is that somehow machine learning systems will become sentient and decide independently from their programmers (and masters) about things that directly affect the livelihood of humans. To some extent, AI already affects the livelihood of humans in an immediate way—creditworthiness is assessed automatically, autopilots mostly navigate vehicles, decisions about whether to grant bail use statistical data as input. More frivolously, we can ask Alexa to switch on the coffee machine.

Fortunately, we are far from a sentient AI system that is ready to manipulate its human creators (or burn their coffee). First, AI systems are engineered, trained and deployed in a specific, goal-oriented manner. While their behavior might give the illusion of general intelligence, it is a com-

bination of rules, heuristics and statistical models that underlie the design. Second, at present tools for *artificial general intelligence* simply do not exist that are able to improve themselves, reason about themselves, and that are able to modify, extend and improve their own architecture while trying to solve general tasks.

A much more pressing concern is how AI is being used in our daily lives. It is likely that many menial tasks fulfilled by truck drivers and shop assistants can and will be automated. Farm robots will likely reduce the cost for organic farming but they will also automate harvesting operations. This phase of the industrial revolution may have profound consequences on large swaths of society (truck drivers and shop assistants are some of the most common jobs in many states). Furthermore, statistical models, when applied without care can lead to racial, gender or age bias and raise reasonable concerns about procedural fairness if automated to drive consequential decisions. It is important to ensure that these algorithms are used with care. With what we know today, this strikes us a much more pressing concern than the potential of malevolent superintelligence to destroy humanity.

Summary

- Machine learning studies how computer systems can leverage *experience* (often data) to improve performance at specific tasks. It combines ideas from statistics, data mining, artificial intelligence, and optimization. Often, it is used as a means of implementing artificially-intelligent solutions.
- As a class of machine learning, representational learning focuses on how to automatically find the appropriate way to represent data. This is often accomplished by a progression of learned transformations.
- Much of the recent progress in deep learning has been triggered by an abundance of data arising from cheap sensors and Internet-scale applications, and by significant progress in computation, mostly through GPUs.
- Whole system optimization is a key component in obtaining good performance. The availability of efficient deep learning frameworks has made design and implementation of this significantly easier.

Exercises

1. Which parts of code that you are currently writing could be "learned", i.e., improved by learning and automatically determining design choices that are made in your code? Does your code include heuristic design choices?
2. Which problems that you encounter have many examples for how to solve them, yet no specific way to automate them? These may be prime candidates for using deep learning.
3. Viewing the development of artificial intelligence as a new industrial revolution, what is the relationship between algorithms and data? Is it similar to steam engines and coal (what is the fundamental difference)?
4. Where else can you apply the end-to-end training approach (such as in Fig. 1.1.2)? Physics? Engineering? Econometrics?

Discussions[39]

[39] https://discuss.d2l.ai/t/22