Dive into Deep Learning

Release 0.7

Aston Zhang, Zack C. Lipton, Mu Li, Alex J. Smola

Jul 26, 2019

CONTENTS

CHAPTER

ONE

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 of us (the authors) entered the field, machine learning didn't 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, 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 increasing autonomy), smart reply systems that anticipate mundane replies, helping people dig out from mountains of email, and software agents that dominate the world's best humans at board games like Go, a feat once deemed to be decades away. Already, these tools are exerting a widening impact, changing the way movies are made, diseases are diagnosed, and playing a growing role in basic sciences – from astrophysics to biology. This book represents our attempt to make deep learning approachable, teaching you both the *concepts*, the *context*, and the *code*.

1.1 About This Book

1.1.1 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 needed 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 full potential of this technology to aid creative entrepreneurs has only been realized over the past ten years, owing to the development of powerful, well-documented frameworks.

Realizing 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, (iv) and the engineering required to train models efficiently, navigating the pitfalls of numerical computing and getting the most out of available hardware. Teaching 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.

We started this book project in July 2017 when we needed to explain MXNet's (then new) Gluon interface to our users. At the time, there were no resources that were simultaneously (1) up to date, (2) covered the full breadth of modern machine learning with anything resembling of technical depth, and (3) interleaved the exposition one expects from an engaging textbook with the clean runnable code one seeks 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.) in the form of blog posts or on GitHub. 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 sporadic topics have been covered in blog posts, e.g. on the website Distill[1] 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 [19], which offers an excellent survey of the concepts behind deep learning, these resources don't 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 (1) be freely available for everyone, (2) offer sufficient technical depth to provide a starting point on the path to actually becoming an applied machine learning scientist, (3) include runnable code, showing readers *how* to solve problems in practice, (4) that allowed for rapid updates, both by us, and also by the community at large, and (5) be complemented by a forum[2] 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 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 17.6. 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.

1.1.2 Learning by Doing

Many textbooks teach a series of topics, each in exhaustive detail. For example, Chris Bishop's excellent textbook [3], 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'll teach most concepts *just in time*. In other words, you'll 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 notebook introduces both a reasonable number of new concepts and provides a single self-contained working example – using a real dataset. 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's 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

[1] http://distill.pub
[2] http://discuss.mxnet.io

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.

Throughout, we'll be working with the MXNet library, which has the rare property of being flexible enough for research while being fast enough for production. 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 `Gluon`'s 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'll often present two versions of the example: one where we implement everything from scratch, relying only on NDArray and automatic differentiation, and another, more practical example, where we write succinct code using `Gluon`. Once we've taught you how some component works, we can just use the `Gluon` version in subsequent tutorials.

1.1.3 Content and Structure

The book can be roughly divided into three parts, which are presented by different colors in Fig. 1.1.1:

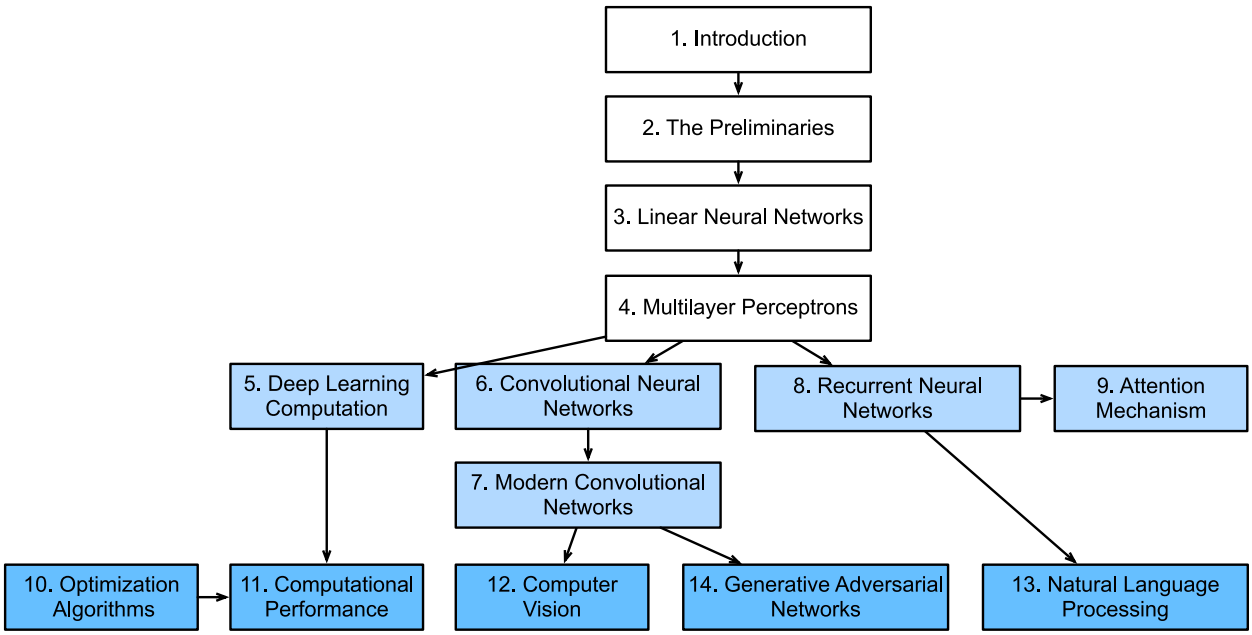

Fig. 1.1.1: Book structure

- The first part covers prerequisites and basics. The first chapter offers an introduction to deep learning in Section 3. In Section 4, we'll quickly bring you up to speed on the prerequisites required for hands-on deep learning, such as how to acquire and run the codes covered in the book. Section 5 and Section 6 cover the most basic concepts and techniques of deep learning, such as linear regression, multi-layer perceptrons and regularization.
- The next four chapters focus on modern deep learning techniques. Section 7 describes the various key components of deep learning calculations and lays the groundwork for the later implementation of more complex models. Next we explain in Section 8 and Section 9, powerful tools that form the backbone of most modern computer vision systems in recent years. Subsequently, we introduce Section 10 models that exploit temporal or sequential structure in data, and are commonly used for natural language processing and time series prediction. Section 11 introduces recent models exploring the attention mechanism. These sections will get you up to speed on the basic tools behind most modern deep learning.

- Part three discusses scalability, efficiency and applications. First, in Section 12, we discuss several common optimization algorithms used to train deep learning models. The next chapter, Section 13 examines several important factors that affect the computational performance of your deep learning code. Section 14 and Section 15 illustrate major applications of deep learning in computer vision and natural language processing, respectively. Finally, Section 16 presents an emerging family of models called generative adversarial networks.

1.1.4 Code

Most sections of this book feature executable code. We recognize 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, our explanations of various techniques might be lacking, sometimes on account of our shortcomings, and equally often on account of the nascent state of the science of deep learning. 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.

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 of such problems, please consult *Installation* (page 7) to update the code and runtime environment.

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 to the d2l package`. For example, these are the packages and modules used by the `d2l` package.

```
# Save to the d2l package
from IPython import display
import collections
import os
import sys
import numpy as np
import math
from matplotlib import pyplot as plt
from mxnet import nd, autograd, gluon, init, context, image
from mxnet.gluon import nn, rnn
import random
import re
import time
import tarfile
import zipfile
```

We give a detailed overview of these functions and classes in Section 17.7.

1.1.5 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'll only assume that you come in with some basics, including (the very basics of) linear algebra, calculus, probability, and Python programming. Moreover, this book's appendix provides 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 [5] covers linear algebra and functional analysis in great depth. All of Statistics [63] is a terrific guide to statistics. And if you have not used Python before, you may want to peruse the Python tutorial[3].

1.1.6 Forum

Associated with this book, we've launched a discussion forum, located at discuss.mxnet.io[4]. When you have questions on any section of the book, you can find the associated discussion page by scanning the QR code at the end of the section to participate in its discussions. The authors of this book and broader MXNet developer community frequently participate in forum discussions.

1.2 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 usernames 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, mstewart141, Mike Müller, NRauschmayr, Prakhar Srivastav, sad-, sfermigier, Sheng Zha, sundeepteki, topecongiro, tpdi, vermicelli, Vishaal Kapoor, vishwesh5, YaYaB, Yuhong Chen, Evgeniy Smirnov, lgov, Simon Corston-Oliver, IgorDzreyev, trunghangx, pmuens, alukovenko, senorcinco, vfdev-5, dsweet, Mohammad Mahdi Rahimi, Abhishek Gupta, uwsd, DomKM, Lisa Oakley, bowen0701, arush15june, prasanth5reddy, brianhendee, mani2106, mtn, lkevinzc, caojilin, Lakshya, Fiete Lüer, Surbhi Vijayvargeeya, Muhyun Kim, dennismalmgren, adursun, Anirudh Dagar, liqingnz, Pedro Larroy, lgov, ati-ozgur, goldmermaid, Jun Wu, Matthias Blume, apeforest, geogunow, jpgard, MaxiBoether, rislam, Leonard Lausen, abhinav-upadhyay, rongruosong, BigBubba, ruslo, rafaelschlatter, liusy182. Moreover, 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.

1.3 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.mxnet.io/.

[3] http://learnpython.org/
[4] https://discuss.mxnet.io/

- Apache MXNet is a powerful library for coding up deep learning models and running them in parallel across GPU cores.
- Gluon is a high level library that makes it easy to code up deep learning models using Apache MXNet.
- Conda is a Python package manager that ensures that all software dependencies are met.
- All notebooks are available for download on GitHub and the conda configurations needed to run this book's code are expressed in the `environment.yml` file.
- If you plan to run this code on GPUs, don't forget to install the necessary drivers and update your configuration.

1.4 Exercises

1. Register an account on the discussion forum of this book discuss.mxnet.io[5].
2. Install Python on your computer.
3. Follow the links at the bottom of the section to the forum, where you'll be able to seek out help and discuss the book and find answers to your questions by engaging the authors and broader community.
4. Create an account on the forum and introduce yourself.

1.5 Scan the QR Code to Discuss[6]

[5] https://discuss.mxnet.io/
[6] https://discuss.mxnet.io/t/2311

CHAPTER

TWO

INSTALLATION

To get you up and running with hands-on experiences, we'll need you to set up with a Python environment, Jupyter's interactive notebooks, the relevant libraries, and the code needed to *run the book*.

2.1 Obtaining Source Codes

The source code package containing all notebooks is available at https://d2l.ai/d2l-en.zip. Please download it and extract it into a folder. For example, on Linux/macos, if you have both `wget` and `unzip` installed, you can do it through:

```
wget https://d2l.ai/d2l-en.zip
unzip d2l-en.zip -d d2l-en
```

2.2 Installing Running Environment

If you have both Python 3.5 or newer and pip installed, the easiest way to install the running environment through pip. Two packages are needed, `d2l` for all dependencies such as Jupyter and saved code blocks, and `mxnet` for deep learning framework we are using. First install `d2l` by

```
pip install d2l
```

If unfortunately something went wrong, please check

1. You are using `pip` for Python 3 instead of Python 2 by checking `pip --version`. If it's Python 2, then you may check if there is a `pip3` available.
2. You are using a recent `pip`, such as version 19. Otherwise you can upgrade it through `pip install --upgrade pip`
3. If you don't have permission to install package in system wide, you can install to your home directory by adding a `--user` flag. Such as `pip install d2l --user`

Before installing `mxnet`, please first check if you are able to access GPUs. If so, please go to *GPU Support* (page 8) for instructions to install a GPU-supported `mxnet`. Otherwise, we can install the CPU version, which is still good enough for the first few chapters.

```
pip install mxnet
```

Once both packages are installed, we now open the Jupyter notebook by

```
jupyter notebook
```

At this point open http://localhost:8888 (which usually opens automatically) in the browser, then you can view and run the code in each section of the book.

2.3 Upgrade to a New Version

Both this book and MXNet are keeping improving. You may want to check a new version from time to time.

1. This URL https://d2l.ai/d2l-en.zip always points to the contents.
2. You can upgrade `d2l` by `pip install d2l -U` or even just install the latest version from Github by `pip install git+https://github.com/d2l-ai/d2l-en`.
3. MXNet can be upgraded by `pip install MXNet -U` as well.

2.4 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[7], you should install a GPU-enabled MXNet.

If you have installed the CPU-only version, then remove it first by

```
pip uninstall mxnet
```

Then you need to find the CUDA version you installed. You may check it through `nvcc --version` or `cat /usr/local/cuda/version.txt`. Assume you have installed CUDA 10.1, then you can install the according MXNet version by

```
pip install mxnet-cu101
```

You may change the last digits according to your CUDA version, e.g. `cu100` for CUDA 10.0 and `cu90` for CUDA 9.0. You can find all available MXNet versions by `pip search mxnet`.

2.5 Exercises

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

2.6 Scan the QR Code to Discuss[8]

[7] https://developer.nvidia.com/cuda-downloads
[8] https://discuss.mxnet.io/t/2315

CHAPTER

THREE

INTRODUCTION

Until recently, nearly all of the computer programs that we interacted with every day were 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 would probably look something like this: (i) users would interact with the application through an interface running in a web browser or mobile application (ii) our application would rely on a commerical database engine to keep track of each user's state and maintain records of all historical transactions (ii) at the heart of our application, running in parallel across many servers, the *business logic* (you might say, the *brains*) would map out in methodical details the appropriate action to 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're able to devise solutions that work 100% of the time, *you should not be using machine learning*.

Fortunately—for the growing community of ML scientists—many problems in automation don't bend so easily to human ingenuity. Imagine huddling around the whiteboard with the smartest minds you know, but this time you are tackling any 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). Machine learning (ML) is the study of powerful techniques that can *learn behavior* from *experience*. As ML algorithm accumulates more experience, typically in the form of observational data or interactions with an environment, their 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's 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.

3.1 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 smartphone 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. 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. 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're stuck, don't worry. We don't know how to write such a program from scratch either. That's why we use ML.

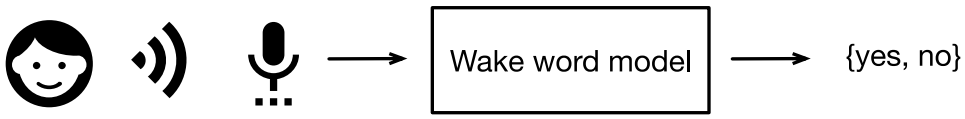

Fig. 3.1.1: Identify an awake word.

Here's the trick. Often, even when we don't 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 don't 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 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*—which, if all goes according to plan, will closely approximate 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'll probably need a model family capable, via another setting of the knobs, of firing `yes` on the word 'Apricot'. We expect that the same model should apply to 'Alexa' recognition and 'Apricot' recognition because these are 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's 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 behaviour from our model.

The training process usually looks like this:

1. Start off with a randomly initialized model that can't 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.

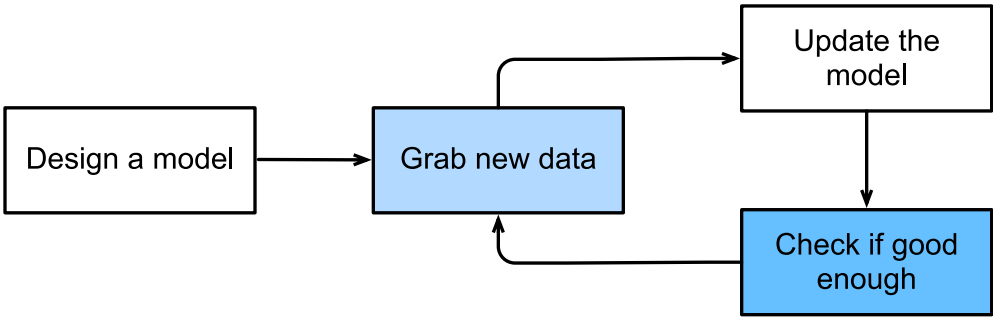

Fig. 3.1.2: A typical training process.

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's a cat, a very large negative number if it's a dog, and something closer to zero if it isn't sure, and this barely scratches the surface of what ML can do.

Deep learning is just one among many popular frameworks for solving machine learning problems. While thus far, we've only talked about machine learning broadly and not deep learning, there's a couple points worth sneaking in here: First, the problems that we've discussed thus far: learning from raw audio signal,

directly from the pixels in images, and mapping between sentences of arbitrary lengths and across languages are problems where deep learning excels and traditional ML tools 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 feature engineering. Secondly, 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.

3.2 The Key Components: Data, Models, and Algorithms

In our *wake-word* example, we described a dataset consisting of audio snippets and binary labels 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* (also called *features* or *covariates*), and examples of both is called *supervised learning*, and it's just one among many *kinds* of machine learning problems. In the next section, we'll 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

3.2.1 Data

It might go without saying that you cannot do data science without data. We could lose hundreds of pages pondering the precise nature of data but for now we'll err on the practical side and focus on the key properties to be concerned with. Generally we are concerned with a collection of *examples* (also called *data points*, *samples*, or *instances*). 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* or *covariates*.

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 corresponding to 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 means we have one less thing to worry about.

However, not all data can easily be represented as fixed length vectors. While we might expect microscrope images to come from standard equipment, we can't expect images mined from the internet to all show up in the same size. While we might imagine cropping images to a standard size, text data resists fixed-length representations even more stubbornly. Consider the product reviews left on e-commerce sites like Amazon 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 either don't work without large data sets. Some others work in the low-data regime, but no better than traditional approaches.

Finally it's 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 well captured by the cliché: *garbage in, garbage out*. Moreover, poor predictive performance isn't 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 doesn't 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 being complicit, or even aware.

3.2.2 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'll discuss some more traditional methods.

3.2.3 Objective functions

Earlier, we introduced machine learning as "learning behavior 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 constitued 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 funcitons 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's 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 doesn't guarantee that we will do well on (unseen) test data. So we'll 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 fails to generalize 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.

3.2.4 Optimization algorithms

Once we've 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.

3.3 Kinds of Machine Learning

In the following sections, we will discuss a few types of machine learning in some more detail. We begin with a list of *objectives*, i.e. a list of things that machine learning can do. Note that the objectives are complemented with a set of techniques of *how* to accomplish them, i.e. training, types of data, etc. The list below is really only sufficient to whet the readers' appetite and to give us a common language when we talk about problems. We will introduce a larger number of such problems as we go along.

3.3.1 Supervised learning

Supervised learning addresses the task of predicting *targets* given input data. The targets, also commonly called *labels*, are generally denoted y. The input data points, also commonly called *examples* or *instances*, are typically denoted $\boldsymbol{x}$. The goal is to produce a model f_θ that maps an input $\boldsymbol{x}$ to a prediction $f_\theta(\boldsymbol{x})$

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 $\boldsymbol{x}$ 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 collection of *labeled examples* $(\boldsymbol{x}_i, y_i)$, where each example $\boldsymbol{x}_i$ is matched up against its correct label.

In probabilistic terms, we typically are interested in estimating the conditional probability $P(y|x)$. While it's just one among several approaches to machine learning, supervised learning accounts for the majority of machine learning in practice. Partly, that's because many important tasks can be described crisply as estimating the probability of some unknown given some available evidence:

- 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 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'll visit many of these problems in depth throughout the first 9 parts of this book.

Put plainly, the learning process looks something like this. Grab a big pile of example inputs, selecting them randomly. Acquire the ground truth labels for each. Together, these inputs and corresponding labels (the desired outputs) comprise the training set. We feed the training dataset into a supervised learning algorithm. So here the *supervised learning algorithm* is a function that takes as input a dataset, and outputs another function, *the learned model*. Then, given a learned model, we can take a new previously unseen input, and predict the corresponding label.

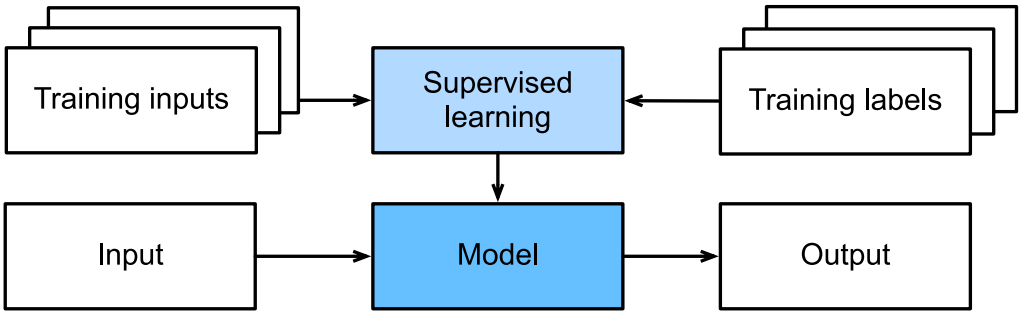

Fig. 3.3.1: Supervised learning.

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. Formally, we call one row in this dataset a *feature vector*, and the object (e.g. a house) it's associated with an *example*.

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 all the classic machine learning problems. We'll typically denote the feature vector for any one example $\mathbf{x_i}$ and the set of feature vectors for all our examples X.

What makes a problem a *regression* is actually the outputs. Say that you're 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*. 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 real values in some range, we call this a regression problem. The goal of our model is to produce predictions (guesses of the price, in our example) that closely approximate the actual target values. We denote these predictions $\hat{y}_i$ and if the notation seems unfamiliar, then just ignore it for now. We'll 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 is a regression problem, and if you designed a great algorithm to accomplish this feat in 2009, you might have won the $1 million Netflix prize[9]. 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's likely, classification, a different fundamental problem type that we'll cover next. Even if you've never worked with machine learning before, you've 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 she sent you a bill of $y_1 = \$350$. Now imagine that your friend hired the same contractor for $x_2 = 2$ hours and that she 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's some base charge and that the contractor then charges per hour. If these assumptions held true, then given these two data points, 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 bias).

In this case, we could produce the parameters that exactly matched the contractor's prices. Sometimes that's not possible, e.g., if some of the variance owes to some factors besides your two features. In these cases, we'll try to learn models that minimize the distance between our predictions and the observed values. In most of our chapters, we'll focus on one of two very common losses, the L1 loss[10] where

$$l(y, y') = \sum_i |y_i - y'_i| \tag{3.3.1}$$

and the least mean squares loss, aka L2 loss[11] where

$$l(y, y') = \sum_i (y_i - y'_i)^2. \tag{3.3.2}$$

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.

Classification

While regression models are great for addressing *how many?* questions, lots of problems don't bend comfortably to this template. For example, a bank wants to add check scanning to their mobile app. This would involve the customer snapping a photo of a check with their smartphone'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 solves is called a classification. It's treated with a distinct set of algorithms than those that are used for regression.

In classification, we want to look at a feature vector, like the pixel values in an image, and then predict which category (formally called *classes*), among some 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,

[9] https://en.wikipedia.org/wiki/Netflix_Prize
[10] http://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.L1Loss
[11] http://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.L2Loss

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'll get into as the book gets more technical, it's pretty hard to optimize a model that can only output a hard categorical assignment, e.g. either *cat* or *dog*. It's a lot easier instead to express the model in the language of probabilities. Given an example x, the 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. This means that we only need $K-1$ numbers to give the probabilities of K categories. This is easy to see for binary classification. If there's a 0.6 (60%) probability that an unfair coin comes up heads, then there's 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 $\Pr(y = \text{cat}|x) = 0.9$. 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 is one notion of confidence. It's not the only notion of confidence and we'll discuss different notions of uncertainty 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 L1 or L2 loss functions, the common loss function for classification problems is called cross-entropy. In MXNet Gluon, the corresponding loss function can be found here[12].

Note that the most likely class is not necessarily the one that you're going to use for your decision. Assume that you find this beautiful mushroom in your backyard:

Fig. 3.3.2: Death cap - do not eat!

[12] https://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.SoftmaxCrossEntropyLoss

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 $\Pr(y = \text{deathcap}|\text{image}) = 0.2$. In other words, the classifier is 80% confident that our mushroom *is not* a death cap. Still, you'd have to be a fool to eat it. That's because the certain benefit of a delicious dinner isn't worth a 20% risk of dying from it. In other words, the effect of the *uncertain risk* by far outweighs the benefit. Let's look at this in math. 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:

$$L(\text{action}|x) = \mathbf{E}_{y \sim p(y|x)}[\text{loss}(\text{action}, y)] \tag{3.3.3}$$

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 - we prefer to misclassify to a related class than to a distant class. Usually, this is referred to as *hierarchical classification*. One early example is due to Linnaeus[13], who organized the animals in a hierarchy.

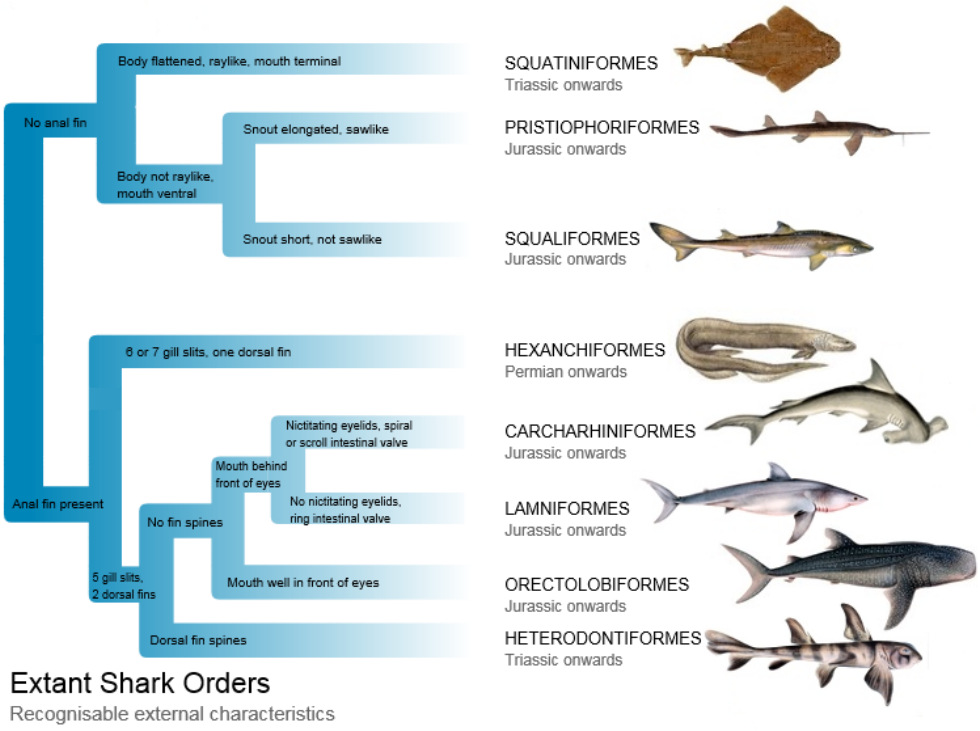

Fig. 3.3.3: Classify sharks

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.

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

Tagging

Some classification problems don't 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.

As you can see, there's 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[14] to do precisely this.

Search and ranking

Sometimes we don't 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 should be displayed for the 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 nonetheless.

One possible solution to this problem is to score every element in the set of possible sets along with a corresponding relevance score and then to retrieve the top-rated elements. PageRank[15] is an early example of such a relevance score. One of the peculiarities is that it didn't depend on the actual query. Instead, it simply helped to order the results that contained the query terms. Nowadays search engines use machine learning and behavioral models to obtain query-dependent relevance scores. There are entire 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 Woody Allen comedies might differ significantly.

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