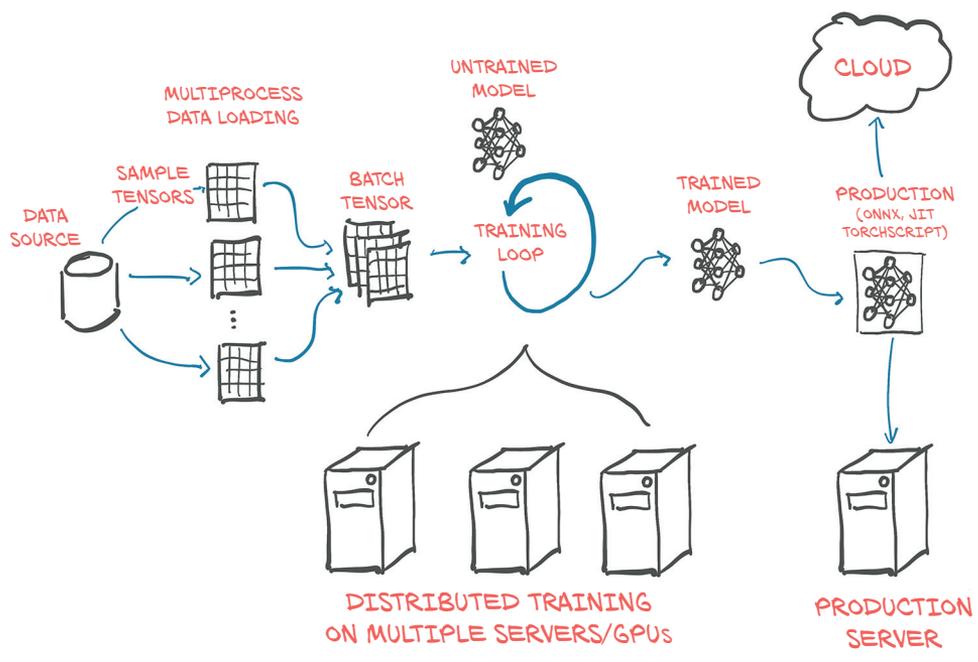


# Deep Learning with PyTorch

Eli Stevens  
Luca Antiga  
Thomas Viehmann  
Foreword by Soumith Chintala





# *Deep Learning with PyTorch*

ELI STEVENS, LUCA ANTIGA,  
AND THOMAS VIEHMANN  
FOREWORD BY SOUMITH CHINTALA



MANNING  
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit [www.manning.com](http://www.manning.com). The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department  
Manning Publications Co.  
20 Baldwin Road  
PO Box 761  
Shelter Island, NY 11964  
Email: [orders@manning.com](mailto:orders@manning.com)

©2020 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

© Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.  
20 Baldwin Road  
PO Box 761  
Shelter Island, NY 11964

Development editor: Frances Lefkowitz  
Technical development editor: Arthur Zubarev  
Review editor: Ivan Martinović  
Production editor: Deirdre Hiam  
Copyeditor: Tiffany Taylor  
Proofreader: Katie Tennant  
Technical proofreader: Kostas Passadis  
Typesetter: Gordan Salinovic  
Cover designer: Marija Tudor

ISBN 9781617295263  
Printed in the United States of America

*To my wife (this book would not have happened without her invaluable support and partnership),  
my parents (I would not have happened without them),  
and my children (this book would have happened a lot sooner but for them).*

*Thank you for being my home, my foundation, and my joy.*

*—Eli Stevens*

*Same :-) But, really, this is for you, Alice and Luigi.*

*—Luca Antiga*

*To Eva, Rebekka, Jonathan, and David.*

*—Thomas Viehmann*



# contents

---

*foreword xv*  
*preface xvii*  
*acknowledgments xix*  
*about this book xxi*  
*about the authors xxvii*  
*about the cover illustration xxviii*

## PART 1 CORE PYTORCH.....1

- 1** *Introducing deep learning and the PyTorch Library 3*
  - 1.1 The deep learning revolution 4
  - 1.2 PyTorch for deep learning 6
  - 1.3 Why PyTorch? 7
    - The deep learning competitive landscape 8*
  - 1.4 An overview of how PyTorch supports deep learning projects 10
  - 1.5 Hardware and software requirements 13
    - Using Jupyter Notebooks 14*
  - 1.6 Exercises 15
  - 1.7 Summary 15

## 2 *Pretrained networks* 16

- 2.1 A pretrained network that recognizes the subject of an image 17
  - Obtaining a pretrained network for image recognition* 19
  - AlexNet* 20 ▪ *ResNet* 22 ▪ *Ready, set, almost run* 22
  - Run!* 25
- 2.2 A pretrained model that fakes it until it makes it 27
  - The GAN game* 28 ▪ *CycleGAN* 29 ▪ *A network that turns horses into zebras* 30
- 2.3 A pretrained network that describes scenes 33
  - NeuralTalk2* 34
- 2.4 Torch Hub 35
- 2.5 Conclusion 37
- 2.6 Exercises 38
- 2.7 Summary 38

## 3 *It starts with a tensor* 39

- 3.1 The world as floating-point numbers 40
- 3.2 Tensors: Multidimensional arrays 42
  - From Python lists to PyTorch tensors* 42 ▪ *Constructing our first tensors* 43 ▪ *The essence of tensors* 43
- 3.3 Indexing tensors 46
- 3.4 Named tensors 46
- 3.5 Tensor element types 50
  - Specifying the numeric type with dtype* 50 ▪ *A dtype for every occasion* 51 ▪ *Managing a tensor's dtype attribute* 51
- 3.6 The tensor API 52
- 3.7 Tensors: Scenic views of storage 53
  - Indexing into storage* 54 ▪ *Modifying stored values: In-place operations* 55
- 3.8 Tensor metadata: Size, offset, and stride 55
  - Views of another tensor's storage* 56 ▪ *Transposing without copying* 58 ▪ *Transposing in higher dimensions* 60
  - Contiguous tensors* 60
- 3.9 Moving tensors to the GPU 62
  - Managing a tensor's device attribute* 63

- 3.10 NumPy interoperability 64
- 3.11 Generalized tensors are tensors, too 65
- 3.12 Serializing tensors 66
  - Serializing to HDF5 with h5py* 67
- 3.13 Conclusion 68
- 3.14 Exercises 68
- 3.15 Summary 68

## 4 *Real-world data representation using tensors* 70

- 4.1 Working with images 71
  - Adding color channels* 72 ▪ *Loading an image file* 72
  - Changing the layout* 73 ▪ *Normalizing the data* 74
- 4.2 3D images: Volumetric data 75
  - Loading a specialized format* 76
- 4.3 Representing tabular data 77
  - Using a real-world dataset* 77 ▪ *Loading a wine data tensor* 78
  - Representing scores* 81 ▪ *One-hot encoding* 81 ▪ *When to categorize* 83 ▪ *Finding thresholds* 84
- 4.4 Working with time series 87
  - Adding a time dimension* 88 ▪ *Shaping the data by time period* 89 ▪ *Ready for training* 90
- 4.5 Representing text 93
  - Converting text to numbers* 94 ▪ *One-hot-encoding characters* 94
  - One-hot encoding whole words* 96 ▪ *Text embeddings* 98
  - Text embeddings as a blueprint* 100
- 4.6 Conclusion 101
- 4.7 Exercises 101
- 4.8 Summary 102

## 5 *The mechanics of learning* 103

- 5.1 A timeless lesson in modeling 104
- 5.2 Learning is just parameter estimation 106
  - A hot problem* 107 ▪ *Gathering some data* 107 ▪ *Visualizing the data* 108 ▪ *Choosing a linear model as a first try* 108
- 5.3 Less loss is what we want 109
  - From problem back to PyTorch* 110

- 5.4 Down along the gradient 113
  - Decreasing loss* 113
  - *Getting analytical* 114
  - *Iterating to fit the model* 116
  - *Normalizing inputs* 119
  - *Visualizing (again)* 122
- 5.5 PyTorch's autograd: Backpropagating all things 123
  - Computing the gradient automatically* 123
  - *Optimizers a la carte* 127
  - *Training, validation, and overfitting* 131
  - Autograd nits and switching it off* 137
- 5.6 Conclusion 139
- 5.7 Exercise 139
- 5.8 Summary 139

## 6 *Using a neural network to fit the data* 141

- 6.1 Artificial neurons 142
  - Composing a multilayer network* 144
  - *Understanding the error function* 144
  - *All we need is activation* 145
  - *More activation functions* 147
  - *Choosing the best activation function* 148
  - What learning means for a neural network* 149
- 6.2 The PyTorch nn module 151
  - Using `__call__` rather than `forward`* 152
  - *Returning to the linear model* 153
- 6.3 Finally a neural network 158
  - Replacing the linear model* 158
  - *Inspecting the parameters* 159
  - Comparing to the linear model* 161
- 6.4 Conclusion 162
- 6.5 Exercises 162
- 6.6 Summary 163

## 7 *Telling birds from airplanes: Learning from images* 164

- 7.1 A dataset of tiny images 165
  - Downloading CIFAR-10* 166
  - *The Dataset class* 166
  - Dataset transforms* 168
  - *Normalizing data* 170
- 7.2 Distinguishing birds from airplanes 172
  - Building the dataset* 173
  - *A fully connected model* 174
  - Output of a classifier* 175
  - *Representing the output as probabilities* 176
  - *A loss for classifying* 180
  - *Training the classifier* 182
  - *The limits of going fully connected* 189
- 7.3 Conclusion 191

7.4 Exercises 191

7.5 Summary 192

## 8 *Using convolutions to generalize* 193

8.1 The case for convolutions 194

*What convolutions do* 194

8.2 Convolutions in action 196

*Padding the boundary* 198 ▪ *Detecting features with convolutions* 200 ▪ *Looking further with depth and pooling* 202  
*Putting it all together for our network* 205

8.3 Subclassing `nn.Module` 207

*Our network as an `nn.Module`* 208 ▪ *How PyTorch keeps track of parameters and submodules* 209 ▪ *The functional API* 210

8.4 Training our convnet 212

*Measuring accuracy* 214 ▪ *Saving and loading our model* 214  
*Training on the GPU* 215

8.5 Model design 217

*Adding memory capacity: Width* 218 ▪ *Helping our model to converge and generalize: Regularization* 219 ▪ *Going deeper to learn more complex structures: Depth* 223 ▪ *Comparing the designs from this section* 228 ▪ *It's already outdated* 229

8.6 Conclusion 229

8.7 Exercises 230

8.8 Summary 231

## PART 2 LEARNING FROM IMAGES IN THE REAL WORLD:

### EARLY DETECTION OF LUNG CANCER.....233

## 9 *Using PyTorch to fight cancer* 235

9.1 Introduction to the use case 236

9.2 Preparing for a large-scale project 237

9.3 What is a CT scan, exactly? 238

9.4 The project: An end-to-end detector for lung cancer 241

*Why can't we just throw data at a neural network until it works?* 245 ▪ *What is a nodule?* 249 ▪ *Our data source: The LUNA Grand Challenge* 251 ▪ *Downloading the LUNA data* 251

9.5 Conclusion 252

9.6 Summary 253

## 10 *Combining data sources into a unified dataset* 254

10.1 Raw CT data files 256

10.2 Parsing LUNA's annotation data 256

*Training and validation sets* 258 ▪ *Unifying our annotation and candidate data* 259

10.3 Loading individual CT scans 262

*Hounsfield Units* 264

10.4 Locating a nodule using the patient coordinate system 265

*The patient coordinate system* 265 ▪ *CT scan shape and voxel sizes* 267 ▪ *Converting between millimeters and voxel addresses* 268 ▪ *Extracting a nodule from a CT scan* 270

10.5 A straightforward dataset implementation 271

*Caching candidate arrays with the `getCtRawCandidate` function* 274 ▪ *Constructing our dataset in `LunaDataset`* `__init__` 275 ▪ *A training/validation split* 275 ▪ *Rendering the data* 277

10.6 Conclusion 277

10.7 Exercises 278

10.8 Summary 278

## 11 *Training a classification model to detect suspected tumors* 279

11.1 A foundational model and training loop 280

11.2 The main entry point for our application 282

11.3 Pretraining setup and initialization 284

*Initializing the model and optimizer* 285 ▪ *Care and feeding of data loaders* 287

11.4 Our first-pass neural network design 289

*The core convolutions* 290 ▪ *The full model* 293

11.5 Training and validating the model 295

*The `computeBatchLoss` function* 297 ▪ *The validation loop is similar* 299

11.6 Outputting performance metrics 300

*The `logMetrics` function* 301

- 11.7 Running the training script 304
  - Needed data for training* 305
  - *Interlude: The enumerateWithEstimate function* 306
- 11.8 Evaluating the model: Getting 99.7% correct means we're done, right? 308
- 11.9 Graphing training metrics with TensorBoard 309
  - Running TensorBoard* 309
  - *Adding TensorBoard support to the metrics logging function* 313
- 11.10 Why isn't the model learning to detect nodules? 315
- 11.11 Conclusion 316
- 11.12 Exercises 316
- 11.13 Summary 316

## 12 *Improving training with metrics and augmentation* 318

- 12.1 High-level plan for improvement 319
- 12.2 Good dogs vs. bad guys: False positives and false negatives 320
- 12.3 Graphing the positives and negatives 322
  - Recall is Roxie's strength* 324
  - *Precision is Preston's forte* 326
  - Implementing precision and recall in logMetrics* 327
  - *Our ultimate performance metric: The F1 score* 328
  - *How does our model perform with our new metrics?* 332
- 12.4 What does an ideal dataset look like? 334
  - Making the data look less like the actual and more like the "ideal"* 336
  - Contrasting training with a balanced LunaDataset to previous runs* 341
  - *Recognizing the symptoms of overfitting* 343
- 12.5 Revisiting the problem of overfitting 345
  - An overfit face-to-age prediction model* 345
- 12.6 Preventing overfitting with data augmentation 346
  - Specific data augmentation techniques* 347
  - *Seeing the improvement from data augmentation* 352
- 12.7 Conclusion 354
- 12.8 Exercises 355
- 12.9 Summary 356

## 13 *Using segmentation to find suspected nodules* 357

- 13.1 Adding a second model to our project 358
- 13.2 Various types of segmentation 360

- 13.3 Semantic segmentation: Per-pixel classification 361
  - The U-Net architecture* 364
- 13.4 Updating the model for segmentation 366
  - Adapting an off-the-shelf model to our project* 367
- 13.5 Updating the dataset for segmentation 369
  - U-Net has very specific input size requirements* 370
  - U-Net trade-offs for 3D vs. 2D data* 370
  - Building the ground truth data* 371
  - Implementing Luna2dSegmentationDataset* 378
  - Designing our training and validation data* 382
  - Implementing TrainingLuna2dSegmentationDataset* 383
  - Augmenting on the GPU* 384
- 13.6 Updating the training script for segmentation 386
  - Initializing our segmentation and augmentation models* 387
  - Using the Adam optimizer* 388
  - Dice loss* 389
  - Getting images into TensorBoard* 392
  - Updating our metrics logging* 396
  - Saving our model* 397
- 13.7 Results 399
- 13.8 Conclusion 401
- 13.9 Exercises 402
- 13.10 Summary 402

## 14 *End-to-end nodule analysis, and where to go next* 404

- 14.1 Towards the finish line 405
- 14.2 Independence of the validation set 407
- 14.3 Bridging CT segmentation and nodule candidate classification 408
  - Segmentation* 410
  - Grouping voxels into nodule candidates* 411
  - Did we find a nodule? Classification to reduce false positives* 412
- 14.4 Quantitative validation 416
- 14.5 Predicting malignancy 417
  - Getting malignancy information* 417
  - An area under the curve baseline: Classifying by diameter* 419
  - Reusing preexisting weights: Fine-tuning* 422
  - More output in TensorBoard* 428
- 14.6 What we see when we diagnose 432
  - Training, validation, and test sets* 433
- 14.7 What next? Additional sources of inspiration (and data) 434
  - Preventing overfitting: Better regularization* 434
  - Refined training data* 437
  - Competition results and research papers* 438

- 14.8 Conclusion 439
  - Behind the curtain* 439
- 14.9 Exercises 441
- 14.10 Summary 441

## PART 3 DEPLOYMENT .....443

### 15 *Deploying to production* 445

- 15.1 Serving PyTorch models 446
  - Our model behind a Flask server* 446 ▪ *What we want from deployment* 448 ▪ *Request batching* 449
- 15.2 Exporting models 455
  - Interoperability beyond PyTorch with ONNX* 455 ▪ *PyTorch's own export: Tracing* 456 ▪ *Our server with a traced model* 458
- 15.3 Interacting with the PyTorch JIT 458
  - What to expect from moving beyond classic Python/PyTorch* 458
  - The dual nature of PyTorch as interface and backend* 460
  - TorchScript* 461 ▪ *Scripting the gaps of traceability* 464
- 15.4 LibTorch: PyTorch in C++ 465
  - Running JITed models from C++* 465 ▪ *C++ from the start: The C++ API* 468
- 15.5 Going mobile 472
  - Improving efficiency: Model design and quantization* 475
- 15.6 Emerging technology: Enterprise serving of PyTorch models 476
- 15.7 Conclusion 477
- 15.8 Exercises 477
- 15.9 Summary 477
  - index* 479



## *foreword*

---

When we started the PyTorch project in mid-2016, we were a band of open source hackers who met online and wanted to write better deep learning software. Two of the three authors of this book, Luca Antiga and Thomas Viehmann, were instrumental in developing PyTorch and making it the success that it is today.

Our goal with PyTorch was to build the most flexible framework possible to express deep learning algorithms. We executed with focus and had a relatively short development time to build a polished product for the developer market. This wouldn't have been possible if we hadn't been standing on the shoulders of giants. PyTorch derives a significant part of its codebase from the Torch7 project started in 2007 by Ronan Collobert and others, which has roots in the Lush programming language pioneered by Yann LeCun and Leon Bottou. This rich history helped us focus on what needed to change, rather than conceptually starting from scratch.

It is hard to attribute the success of PyTorch to a single factor. The project offers a good user experience and enhanced debuggability and flexibility, ultimately making users more productive. The huge adoption of PyTorch has resulted in a beautiful ecosystem of software and research built on top of it, making PyTorch even richer in its experience.

Several courses and university curricula, as well as a huge number of online blogs and tutorials, have been offered to make PyTorch easier to learn. However, we have seen very few books. In 2017, when someone asked me, "When is the PyTorch book going to be written?" I responded, "If it gets written now, I can guarantee that it will be outdated by the time it is completed."

With the publication of *Deep Learning with PyTorch*, we finally have a definitive treatise on PyTorch. It covers the basics and abstractions in great detail, tearing apart the underpinnings of data structures like tensors and neural networks and making sure you understand their implementation. Additionally, it covers advanced subjects such as JIT and deployment to production (an aspect of PyTorch that no other book currently covers).

Additionally, the book covers applications, taking you through the steps of using neural networks to help solve a complex and important medical problem. With Luca's deep expertise in bioengineering and medical imaging, Eli's practical experience creating software for medical devices and detection, and Thomas's background as a PyTorch core developer, this journey is treated carefully, as it should be.

All in all, I hope this book becomes your "extended" reference document and an important part of your library or workshop.

SOUMITH CHINTALA  
COCREATOR OF PYTORCH

## *preface*

---

As kids in the 1980s, taking our first steps on our Commodore VIC 20 (Eli), the Sinclair Spectrum 48K (Luca), and the Commodore C16 (Thomas), we saw the dawn of personal computers, learned to code and write algorithms on ever-faster machines, and often dreamed about where computers would take us. We also were painfully aware of the gap between what computers did in movies and what they could do in real life, collectively rolling our eyes when the main character in a spy movie said, “Computer, enhance.”

Later on, during our professional lives, two of us, Eli and Luca, independently challenged ourselves with medical image analysis, facing the same kind of struggle when writing algorithms that could handle the natural variability of the human body. There was a lot of heuristics involved when choosing the best mix of algorithms that could make things work and save the day. Thomas studied neural nets and pattern recognition at the turn of the century but went on to get a PhD in mathematics doing modeling.

When deep learning came about at the beginning of the 2010s, making its initial appearance in computer vision, it started being applied to medical image analysis tasks like the identification of structures or lesions on medical images. It was at that time, in the first half of the decade, that deep learning appeared on our individual radars. It took a bit to realize that deep learning represented a whole new way of writing software: a new class of multipurpose algorithms that could learn how to solve complicated tasks through the observation of data.

To our kids-of-the-80s minds, the horizon of what computers could do expanded overnight, limited not by the brains of the best programmers, but by the data, the neural network architecture, and the training process. The next step was getting our hands dirty. Luca chose Torch 7 (<http://torch.ch>), a venerable precursor to PyTorch; it's nimble, lightweight, and fast, with approachable source code written in Lua and plain C, a supportive community, and a long history behind it. For Luca, it was love at first sight. The only real drawback with Torch 7 was being detached from the ever-growing Python data science ecosystem that the other frameworks could draw from. Eli had been interested in AI since college,<sup>1</sup> but his career pointed him in other directions, and he found other, earlier deep learning frameworks a bit too laborious to get enthusiastic about using them for a hobby project.

So we all got really excited when the first PyTorch release was made public on January 18, 2017. Luca started contributing to the core, and Eli was part of the community very early on, submitting the odd bug fix, feature, or documentation update. Thomas contributed a ton of features and bug fixes to PyTorch and eventually became one of the independent core contributors. There was the feeling that something big was starting up, at the right level of complexity and with a minimal amount of cognitive overhead. The lean design lessons learned from the Torch 7 days were being carried over, but this time with a modern set of features like automatic differentiation, dynamic computation graphs, and NumPy integration.

Given our involvement and enthusiasm, and after organizing a couple of PyTorch workshops, writing a book felt like a natural next step. The goal was to write a book that would have been appealing to our former selves getting started just a few years back.

Predictably, we started with grandiose ideas: teach the basics, walk through end-to-end projects, and demonstrate the latest and greatest models in PyTorch. We soon realized that would take a lot more than a single book, so we decided to focus on our initial mission: devote time and depth to cover the key concepts underlying PyTorch, assuming little or no prior knowledge of deep learning, and get to the point where we could walk our readers through a complete project. For the latter, we went back to our roots and chose to demonstrate a medical image analysis challenge.

---

<sup>1</sup> Back when “deep” neural networks meant *three* hidden layers!

## *acknowledgments*

---

We are deeply indebted to the PyTorch team. It is through their collective effort that PyTorch grew organically from a summer internship project to a world-class deep learning tool. We would like to mention Soumith Chintala and Adam Paszke, who, in addition to their technical excellence, worked actively toward adopting a “community first” approach to managing the project. The level of health and inclusiveness in the PyTorch community is a testament to their actions.

Speaking of community, PyTorch would not be what it is if not for the relentless work of individuals helping early adopters and experts alike on the discussion forum. Of all the honorable contributors, Piotr Bialecki deserves our particular badge of gratitude. Speaking of the book, a particular shout-out goes to Joe Spisak, who believed in the value that this book could bring to the community, and also Jeff Smith, who did an incredible amount of work to bring that value to fruition. Bruce Lin’s work to excerpt part 1 of this text and provide it to the PyTorch community free of charge is also hugely appreciated.

We would like to thank the team at Manning for guiding us through this journey, always aware of the delicate balance between family, job, and writing in our respective lives. Thanks to Erin Twohey for reaching out and asking if we’d be interested in writing a book, and thanks to Michael Stephens for tricking us into saying yes. We *told* you we had no time! Brian Hanafée went above and beyond a reviewer’s duty. Arthur Zubarev and Kostas Passadis gave great feedback, and Jennifer Houle had to deal with our wacky art style. Our copyeditor, Tiffany Taylor, has an impressive eye for detail; any mistakes are ours and ours alone. We would also like to thank our project editor,

Deirdre Hiam, our proofreader, Katie Tennant, and our review editor, Ivan Martinić. There are also a host of people working behind the scenes, glimpsed only on the CC list of status update threads, and all necessary to bring this book to print. Thank you to every name we've left off this list! The anonymous reviewers who gave their honest feedback helped make this book what it is.

Frances Lefkowitz, our tireless editor, deserves a medal and a week on a tropical island after dragging this book over the finish line. Thank you for all you've done and for the grace with which you did it.

We would also like to thank our reviewers, who have helped to improve our book in many ways: Aleksandr Erofeev, Audrey Carstensen, Bachir Chihani, Carlos Andres Mariscal, Dale Neal, Daniel Berecz, Doniyor Ulmasov, Ezra Stevens, Godfred Asamoah, Helen Mary Labao Barrameda, Hilde Van Gysel, Jason Leonard, Jeff Coggshall, Kostas Passadis, Linnsey Nil, Mathieu Zhang, Michael Constant, Miguel Montalvo, Orlando Alejo Méndez Morales, Philippe Van Bergen, Reece Stevens, Srinivas K. Raman, and Yujan Shrestha.

To our friends and family, wondering what rock we've been hiding under these past two years: Hi! We missed you! Let's have dinner sometime.

## *about this book*

---

This book has the aim of providing the foundations of deep learning with PyTorch and showing them in action in a real-life project. We strive to provide the key concepts underlying deep learning and show how PyTorch puts them in the hands of practitioners. In the book, we try to provide intuition that will support further exploration, and in doing so we selectively delve into details to show what is going on behind the curtain.

*Deep Learning with PyTorch* doesn't try to be a reference book; rather, it's a conceptual companion that will allow you to independently explore more advanced material online. As such, we focus on a subset of the features offered by PyTorch. The most notable absence is recurrent neural networks, but the same is true for other parts of the PyTorch API.

### ***Who should read this book***

This book is meant for developers who are or aim to become deep learning practitioners and who want to get acquainted with PyTorch. We imagine our typical reader to be a computer scientist, data scientist, or software engineer, or an undergraduate-or-later student in a related program. Since we don't assume prior knowledge of deep learning, some parts in the first half of the book may be a repetition of concepts that are already known to experienced practitioners. For those readers, we hope the exposition will provide a slightly different angle to known topics.

We expect readers to have basic knowledge of imperative and object-oriented programming. Since the book uses Python, you should be familiar with the syntax and operating environment. Knowing how to install Python packages and run scripts on

your platform of choice is a prerequisite. Readers coming from C++, Java, JavaScript, Ruby, or other such languages should have an easy time picking it up but will need to do some catch-up outside this book. Similarly, being familiar with NumPy will be useful, if not strictly required. We also expect familiarity with some basic linear algebra, such as knowing what matrices and vectors are and what a dot product is.

### **How this book is organized: A roadmap**

*Deep Learning with PyTorch* is organized in three distinct parts. Part 1 covers the foundations, while part 2 walks you through an end-to-end project, building on the basic concepts introduced in part 1 and adding more advanced ones. The short part 3 rounds off the book with a tour of what PyTorch offers for deployment. You will likely notice different voices and graphical styles among the parts. Although the book is a result of endless hours of collaborative planning, discussion, and editing, the act of writing and authoring graphics was split among the parts: Luca was primarily in charge of part 1 and Eli of part 2.<sup>2</sup> When Thomas came along, he tried to blend the style in part 3 and various sections here and there with the writing in parts 1 and 2. Rather than finding a minimum common denominator, we decided to preserve the original voices that characterized the parts.

Following is a breakdown of each part into chapters and a brief description of each.

#### **PART 1**

In part 1, we take our first steps with PyTorch, building the fundamental skills needed to understand PyTorch projects out there in the wild as well as starting to build our own. We'll cover the PyTorch API and some behind-the-scenes features that make PyTorch the library it is, and work on training an initial classification model. By the end of part 1, we'll be ready to tackle a real-world project.

Chapter 1 introduces PyTorch as a library and its place in the deep learning revolution, and touches on what sets PyTorch apart from other deep learning frameworks.

Chapter 2 shows PyTorch in action by running examples of pretrained networks; it demonstrates how to download and run models in PyTorch Hub.

Chapter 3 introduces the basic building block of PyTorch—the tensor—showing its API and going behind the scenes with some implementation details.

Chapter 4 demonstrates how different kinds of data can be represented as tensors and how deep learning models expects tensors to be shaped.

Chapter 5 walks through the mechanics of learning through gradient descent and how PyTorch enables it with automatic differentiation.

Chapter 6 shows the process of building and training a neural network for regression in PyTorch using the `nn` and `optim` modules.

Chapter 7 builds on the previous chapter to create a fully connected model for image classification and expand the knowledge of the PyTorch API.

Chapter 8 introduces convolutional neural networks and touches on more advanced concepts for building neural network models and their PyTorch implementation.

---

<sup>2</sup> A smattering of Eli's and Thomas's art appears in other parts; don't be shocked if the style changes mid-chapter!

**PART 2**

In part 2, each chapter moves us closer to a comprehensive solution to automatic detection of lung cancer. We'll use this difficult problem as motivation to demonstrate the real-world approaches needed to solve large-scale problems like cancer screening. It is a large project with a focus on clean engineering, troubleshooting, and problem solving.

Chapter 9 describes the end-to-end strategy we'll use for lung tumor classification, starting from computed tomography (CT) imaging.

Chapter 10 loads the human annotation data along with the images from CT scans and converts the relevant information into tensors, using standard PyTorch APIs.

Chapter 11 introduces a first classification model that consumes the training data introduced in chapter 10. We train the model and collect basic performance metrics. We also introduce using TensorBoard to monitor training.

Chapter 12 explores and implements standard performance metrics and uses those metrics to identify weaknesses in the training done previously. We then mitigate those flaws with an improved training set that uses data balancing and augmentation.

Chapter 13 describes segmentation, a pixel-to-pixel model architecture that we use to produce a heatmap of possible nodule locations that covers the entire CT scan. This heatmap can be used to find nodules on CT scans for which we do not have human-annotated data.

Chapter 14 implements the final end-to-end project: diagnosis of cancer patients using our new segmentation model followed by classification.

**PART 3**

Part 3 is a single chapter on deployment. Chapter 15 provides an overview of how to deploy PyTorch models to a simple web service, embed them in a C++ program, or bring them to a mobile phone.

**About the code**

All of the code in this book was written for Python 3.6 or later. The code for the book is available for download from Manning's website ([www.manning.com/books/deep-learning-with-pytorch](http://www.manning.com/books/deep-learning-with-pytorch)) and on GitHub (<https://github.com/deep-learning-with-pytorch/dlwpt-code>). Version 3.6.8 was current at the time of writing and is what we used to test the examples in this book. For example:

```
$ python
Python 3.6.8 (default, Jan 14 2019, 11:02:34)
[GCC 8.0.1 20180414 on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Command lines intended to be entered at a Bash prompt start with \$ (for example, the \$ python line in this example). Fixed-width inline code looks like self.

Code blocks that begin with >>> are transcripts of a session at the Python interactive prompt. The >>> characters are not meant to be considered input; text lines that

do not start with `>>>` or `...` are output. In some cases, an extra blank line is inserted before the `>>>` to improve readability in print. These blank lines are not included when you actually enter the text at the interactive prompt:

```
>>> print("Hello, world!")
Hello, world!

>>> print("Until next time...")
Until next time...
```

This blank line would not be  
present during an actual  
interactive session.

We also make heavy use of Jupyter Notebooks, as described in chapter 1, in section 1.5.1. Code from a notebook that we provide as part of the official GitHub repository looks like this:

```
# In[1]:
print("Hello, world!")

# Out[1]:
Hello, world!

# In[2]:
print("Until next time...")

# Out[2]:
Until next time...
```

Almost all of our example notebooks contain the following boilerplate in the first cell (some lines may be missing in early chapters), which we skip including in the book after this point:

```
# In[1]:
%matplotlib inline
from matplotlib import pyplot as plt
import numpy as np

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

torch.set_printoptions(edgeitems=2)
torch.manual_seed(123)
```

Otherwise, code blocks are partial or entire sections of `.py` source files.

#### Listing 15.1 `main.py:5, def main`

```
def main():
    print("Hello, world!")

if __name__ == '__main__':
    main()
```

Many of the code samples in the book are presented with two-space indents. Due to the limitations of print, code listings are limited to 80-character lines, which can be impractical for heavily indented sections of code. The use of two-space indents helps to mitigate the excessive line wrapping that would otherwise be present. All of the code available for download for the book (again, at [www.manning.com/books/deep-learning-with-pytorch](http://www.manning.com/books/deep-learning-with-pytorch) and <https://github.com/deep-learning-with-pytorch/dlwpt-code>) uses a consistent four-space indent. Variables named with a `_t` suffix are tensors stored in CPU memory, `_g` are tensors in GPU memory, and `_a` are NumPy arrays.

### **Hardware and software requirements**

Part 1 has been designed to not require any particular computing resources. Any recent computer or online computing resource will be adequate. Similarly, no certain operating system is required. In part 2, we anticipate that completing a full training run for the more advanced examples will require a CUDA-capable GPU. The default parameters used in part 2 assume a GPU with 8 GB of RAM (we suggest an NVIDIA GTX 1070 or better), but the parameters can be adjusted if your hardware has less RAM available. The raw data needed for part 2's cancer-detection project is about 60 GB to download, and you will need a total of 200 GB (at minimum) of free disk space on the system that will be used for training. Luckily, online computing services recently started offering GPU time for free. We discuss computing requirements in more detail in the appropriate sections.

You need Python 3.6 or later; instructions can be found on the Python website ([www.python.org/downloads](http://www.python.org/downloads)). For PyTorch installation information, see the Get Started guide on the official PyTorch website (<https://pytorch.org/get-started/locally>). We suggest that Windows users install with Anaconda or Miniconda (<https://www.anaconda.com/distribution> or <https://docs.conda.io/en/latest/miniconda.html>). Other operating systems like Linux typically have a wider variety of workable options, with Pip being the most common package manager for Python. We provide a `requirements.txt` file that Pip can use to install dependencies. Since current Apple laptops do not include GPUs that support CUDA, the precompiled macOS packages for PyTorch are CPU-only. Of course, experienced users are free to install packages in the way that is most compatible with your preferred development environment.

### **liveBook discussion forum**

Purchase of *Deep Learning with PyTorch* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the forum, go to <https://livebook.manning.com/#!/book/deep-learning-with-pytorch/discussion>. You can learn more about Manning's forums and the rules of conduct at <https://livebook.manning.com/#!/discussion>. Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific

amount of participation on the part of the authors, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking them some challenging questions lest their interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

### ***Other online resources***

Although this book does not assume prior knowledge of deep learning, it is not a foundational introduction to deep learning. We cover the basics, but our focus is on proficiency with the PyTorch library. We encourage interested readers to build up an intuitive understanding of deep learning either before, during, or after reading this book. Toward that end, *Grokking Deep Learning* ([www.manning.com/books/grokking-deep-learning](http://www.manning.com/books/grokking-deep-learning)) is a great resource for developing a strong mental model and intuition about the mechanism underlying deep neural networks. For a thorough introduction and reference, we direct you to *Deep Learning* by Goodfellow et al. ([www.deeplearningbook.org](http://www.deeplearningbook.org)). And of course, Manning Publications has an extensive catalog of deep learning titles ([www.manning.com/catalog#section-83](http://www.manning.com/catalog#section-83)) that cover a wide variety of topics in the space. Depending on your interests, many of them will make an excellent next book to read.

## *about the authors*

---

Eli Stevens has spent the majority of his career working at startups in Silicon Valley, with roles ranging from software engineer (making enterprise networking appliances) to CTO (developing software for radiation oncology). At publication, he is working on machine learning in the self-driving-car industry.

Luca Antiga worked as a researcher in biomedical engineering in the 2000s, and spent the last decade as a cofounder and CTO of an AI engineering company. He has contributed to several open source projects, including the PyTorch core. He recently cofounded a US-based startup focused on infrastructure for data-defined software.

Thomas Viehmann is a machine learning and PyTorch specialty trainer and consultant based in Munich, Germany, and a PyTorch core developer. With a PhD in mathematics, he is not scared by theory, but he is thoroughly practical when applying it to computing challenges.

## *about the cover illustration*

---

The figure on the cover of *Deep Learning with PyTorch* is captioned “Kardinian.” The illustration is taken from a collection of dress costumes from various countries by Jacques Grasset de Saint-Sauveur (1757-1810), titled *Costumes civils actuels de tous les peuples connus*, published in France in 1788. Each illustration is finely drawn and colored by hand. The rich variety of Grasset de Saint-Sauveur’s collection reminds us vividly of how culturally apart the world’s towns and regions were just 200 years ago. Isolated from each other, people spoke different dialects and languages. In the streets or in the countryside, it was easy to identify where they lived and what their trade or station in life was just by their dress.

The way we dress has changed since then and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns, regions, or countries. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Grasset de Saint-Sauveur’s pictures.

# *Part 1*

## *Core PyTorch*

**W**elcome to the first part of this book. This is where we'll take our first steps with PyTorch, gaining the fundamental skills needed to understand its anatomy and work out the mechanics of a PyTorch project.

In chapter 1, we'll make our first contact with PyTorch, understand what it is and what problems it solves, and how it relates to other deep learning frameworks. Chapter 2 will take us on a tour, giving us a chance to play with models that have been pretrained on fun tasks. Chapter 3 gets a bit more serious and teaches the basic data structure used in PyTorch programs: the tensor. Chapter 4 will take us on another tour, this time across ways to represent data from different domains as PyTorch tensors. Chapter 5 unveils how a program can learn from examples and how PyTorch supports this process. Chapter 6 provides the fundamentals of what a neural network is and how to build a neural network with PyTorch. Chapter 7 tackles a simple image classification problem with a neural network architecture. Finally, chapter 8 shows how the same problem can be cracked in a much smarter way using a convolutional neural network.

By the end of part 1, we'll have what it takes to tackle a real-world problem with PyTorch in part 2.



# 1

## *Introducing deep learning and the PyTorch Library*

---

### ***This chapter covers***

- How deep learning changes our approach to machine learning
- Understanding why PyTorch is a good fit for deep learning
- Examining a typical deep learning project
- The hardware you'll need to follow along with the examples

The poorly defined term *artificial intelligence* covers a set of disciplines that have been subjected to a tremendous amount of research, scrutiny, confusion, fantastical hype, and sci-fi fearmongering. Reality is, of course, far more sanguine. It would be disingenuous to assert that today's machines are learning to “think” in any human sense of the word. Rather, we've discovered a general class of algorithms

that are able to approximate complicated, nonlinear processes very, very effectively, which we can use to automate tasks that were previously limited to humans.

For example, at <https://talktotransformer.com>, a language model called GPT-2 can generate coherent paragraphs of text one word at a time. When we fed it this very paragraph, it produced the following:

*Next we're going to feed in a list of phrases from a corpus of email addresses, and see if the program can parse the lists as sentences. Again, this is much more complicated and far more complex than the search at the beginning of this post, but hopefully helps you understand the basics of constructing sentence structures in various programming languages.*

That's remarkably coherent for a machine, even if there isn't a well-defined thesis behind the rambling.

Even more impressively, the ability to perform these formerly human-only tasks is acquired *through examples*, rather than encoded by a human as a set of handcrafted rules. In a way, we're learning that intelligence is a notion we often conflate with self-awareness, and self-awareness is definitely not required to successfully carry out these kinds of tasks. In the end, the question of computer intelligence might not even be important. Edsger W. Dijkstra found that the question of whether machines could think was "about as relevant as the question of whether Submarines Can Swim."<sup>1</sup>

That general class of algorithms we're talking about falls under the AI subcategory of *deep learning*, which deals with training mathematical entities named *deep neural networks* by presenting instructive examples. Deep learning uses large amounts of data to approximate complex functions whose inputs and outputs are far apart, like an input image and, as output, a line of text describing the input; or a written script as input and a natural-sounding voice reciting the script as output; or, even more simply, associating an image of a golden retriever with a flag that tells us "Yes, a golden retriever is present." This kind of capability allows us to create programs with functionality that was, until very recently, exclusively the domain of human beings.

## 1.1 *The deep learning revolution*

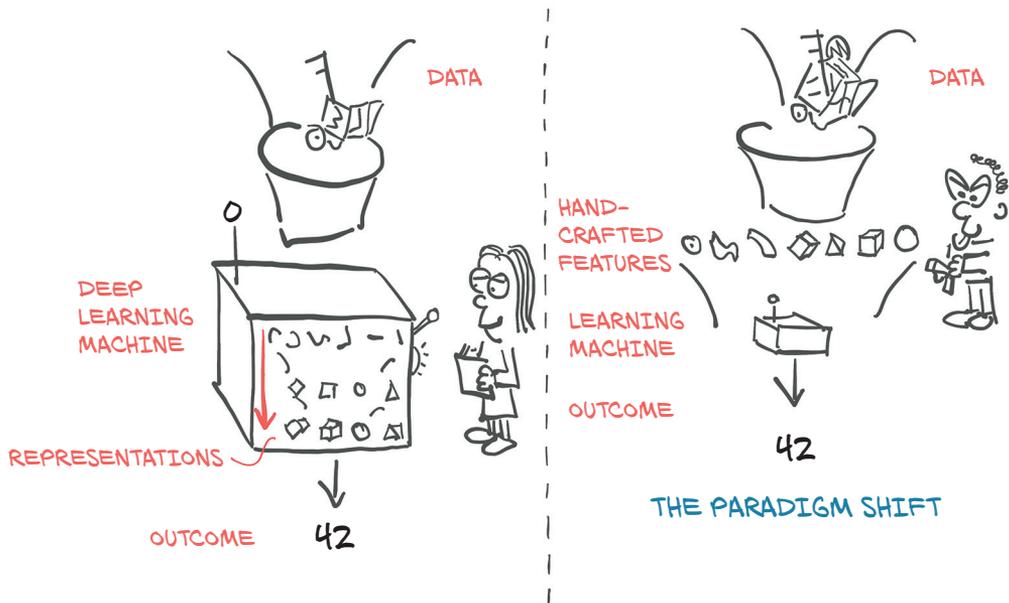
To appreciate the paradigm shift ushered in by this deep learning approach, let's take a step back for a bit of perspective. Until the last decade, the broader class of systems that fell under the label *machine learning* relied heavily on *feature engineering*. Features are transformations on input data that facilitate a downstream algorithm, like a classifier, to produce correct outcomes on new data. Feature engineering consists of coming up with the right transformations so that the downstream algorithm can solve a task. For instance, in order to tell ones from zeros in images of handwritten digits, we would come up with a set of filters to estimate the direction of edges over the image, and then train a classifier to predict the correct digit given a distribution of edge directions. Another useful feature could be the number of enclosed holes, as seen in a zero, an eight, and, particularly, loopy twos.

---

<sup>1</sup> Edsger W. Dijkstra, "The Threats to Computing Science," <http://mng.bz/nPJ5>.

Deep learning, on the other hand, deals with finding such representations automatically, from raw data, in order to successfully perform a task. In the ones versus zeros example, filters would be refined during training by iteratively looking at pairs of examples and target labels. This is not to say that feature engineering has no place with deep learning; we often need to inject some form of prior knowledge in a learning system. However, the ability of a neural network to ingest data and extract useful representations on the basis of examples is what makes deep learning so powerful. The focus of deep learning practitioners is not so much on handcrafting those representations, but on operating on a mathematical entity so that it discovers representations from the training data autonomously. Often, these automatically created features are better than those that are handcrafted! As with many disruptive technologies, this fact has led to a change in perspective.

On the left side of figure 1.1, we see a practitioner busy defining engineering features and feeding them to a learning algorithm; the results on the task will be as good as the features the practitioner engineers. On the right, with deep learning, the raw data is fed to an algorithm that extracts hierarchical features automatically, guided by the optimization of its own performance on the task; the results will be as good as the ability of the practitioner to drive the algorithm toward its goal.



**Figure 1.1** Deep learning exchanges the need to handcraft features for an increase in data and computational requirements.

Starting from the right side in figure 1.1, we already get a glimpse of what we need to execute successful deep learning:

- We need a way to ingest whatever data we have at hand.
- We somehow need to define the deep learning machine.
- We must have an automated way, *training*, to obtain useful representations and make the machine produce desired outputs.

This leaves us with taking a closer look at this training thing we keep talking about. During training, we use a *criterion*, a real-valued function of model outputs and reference data, to provide a numerical score for the discrepancy between the desired and actual output of our model (by convention, a lower score is typically better). Training consists of driving the criterion toward lower and lower scores by incrementally modifying our deep learning machine until it achieves low scores, even on data not seen during training.

## 1.2 *PyTorch for deep learning*

PyTorch is a library for Python programs that facilitates building deep learning projects. It emphasizes flexibility and allows deep learning models to be expressed in idiomatic Python. This approachability and ease of use found early adopters in the research community, and in the years since its first release, it has grown into one of the most prominent deep learning tools across a broad range of applications.

As Python does for programming, PyTorch provides an excellent introduction to deep learning. At the same time, PyTorch has been proven to be fully qualified for use in professional contexts for real-world, high-profile work. We believe that PyTorch's clear syntax, streamlined API, and easy debugging make it an excellent choice for introducing deep learning. We highly recommend studying PyTorch for your first deep learning library. Whether it ought to be the last deep learning library you learn is a decision we leave up to you.

At its core, the deep learning machine in figure 1.1 is a rather complex mathematical function mapping inputs to an output. To facilitate expressing this function, PyTorch provides a core data structure, the *tensor*, which is a multidimensional array that shares many similarities with NumPy arrays. Around that foundation, PyTorch comes with features to perform accelerated mathematical operations on dedicated hardware, which makes it convenient to design neural network architectures and train them on individual machines or parallel computing resources.

This book is intended as a starting point for software engineers, data scientists, and motivated students fluent in Python to become comfortable using PyTorch to build deep learning projects. We want this book to be as accessible and useful as possible, and we expect that you will be able to take the concepts in this book and apply them to other domains. To that end, we use a hands-on approach and encourage you to keep your computer at the ready, so you can play with the examples and take them a step further. By the time we are through with the book, we expect you to be able to

take a data source and build out a deep learning project with it, supported by the excellent official documentation.

Although we stress the practical aspects of building deep learning systems with PyTorch, we believe that providing an accessible introduction to a foundational deep learning tool is more than just a way to facilitate the acquisition of new technical skills. It is a step toward equipping a new generation of scientists, engineers, and practitioners from a wide range of disciplines with working knowledge that will be the backbone of many software projects during the decades to come.

In order to get the most out of this book, you will need two things:

- Some experience programming in Python. We're not going to pull any punches on that one; you'll need to be up on Python data types, classes, floating-point numbers, and the like.
- A willingness to dive in and get your hands dirty. We'll be starting from the basics and building up our working knowledge, and it will be much easier for you to learn if you follow along with us.

*Deep Learning with PyTorch* is organized in three distinct parts. Part 1 covers the foundations, examining in detail the facilities PyTorch offers to put the sketch of deep learning in figure 1.1 into action with code. Part 2 walks you through an end-to-end project involving medical imaging: finding and classifying tumors in CT scans, building on the basic concepts introduced in part 1, and adding more advanced topics. The short part 3 rounds off the book with a tour of what PyTorch offers for deploying deep learning models to production.

Deep learning is a huge space. In this book, we will be covering a tiny part of that space: specifically, using PyTorch for smaller-scope classification and segmentation projects, with image processing of 2D and 3D datasets used for most of the motivating examples. This book focuses on practical PyTorch, with the aim of covering enough ground to allow you to solve real-world machine learning problems, such as in vision, with deep learning or explore new models as they pop up in research literature. Most, if not all, of the latest publications related to deep learning research can be found in the arXiv public preprint repository, hosted at <https://arxiv.org>.<sup>2</sup>

### 1.3 Why PyTorch?

As we've said, deep learning allows us to carry out a very wide range of complicated tasks, like machine translation, playing strategy games, or identifying objects in cluttered scenes, by exposing our model to illustrative examples. In order to do so in practice, we need tools that are flexible, so they can be adapted to such a wide range of problems, and efficient, to allow training to occur over large amounts of data in reasonable times; and we need the trained model to perform correctly in the presence of variability in the inputs. Let's take a look at some of the reasons we decided to use PyTorch.

---

<sup>2</sup> We also recommend [www.arxiv-sanity.com](http://www.arxiv-sanity.com) to help organize research papers of interest.

PyTorch is easy to recommend because of its simplicity. Many researchers and practitioners find it easy to learn, use, extend, and debug. It's Pythonic, and while like any complicated domain it has caveats and best practices, using the library generally feels familiar to developers who have used Python previously.

More concretely, programming the deep learning machine is very natural in PyTorch. PyTorch gives us a data type, the `Tensor`, to hold numbers, vectors, matrices, or arrays in general. In addition, it provides functions for operating on them. We can program with them incrementally and, if we want, interactively, just like we are used to from Python. If you know NumPy, this will be very familiar.

But PyTorch offers two things that make it particularly relevant for deep learning: first, it provides accelerated computation using graphical processing units (GPUs), often yielding speedups in the range of 50x over doing the same calculation on a CPU. Second, PyTorch provides facilities that support numerical optimization on generic mathematical expressions, which deep learning uses for training. Note that both features are useful for scientific computing in general, not exclusively for deep learning. In fact, we can safely characterize PyTorch as a high-performance library with optimization support for scientific computing in Python.

A design driver for PyTorch is expressivity, allowing a developer to implement complicated models without undue complexity being imposed by the library (it's not a framework!). PyTorch arguably offers one of the most seamless translations of ideas into Python code in the deep learning landscape. For this reason, PyTorch has seen widespread adoption in research, as witnessed by the high citation counts at international conferences.<sup>3</sup>

PyTorch also has a compelling story for the transition from research and development into production. While it was initially focused on research workflows, PyTorch has been equipped with a high-performance C++ runtime that can be used to deploy models for inference without relying on Python, and can be used for designing and training models in C++. It has also grown bindings to other languages and an interface for deploying to mobile devices. These features allow us to take advantage of PyTorch's flexibility and at the same time take our applications where a full Python runtime would be hard to get or would impose expensive overhead.

Of course, claims of ease of use and high performance are trivial to make. We hope that by the time you are in the thick of this book, you'll agree with us that our claims here are well founded.

### **1.3.1** *The deep learning competitive landscape*

While all analogies are flawed, it seems that the release of PyTorch 0.1 in January 2017 marked the transition from a Cambrian-explosion-like proliferation of deep learning libraries, wrappers, and data-exchange formats into an era of consolidation and unification.

---

<sup>3</sup> At the International Conference on Learning Representations (ICLR) 2019, PyTorch appeared as a citation in 252 papers, up from 87 the previous year and at the same level as TensorFlow, which appeared in 266 papers.

**NOTE** The deep learning landscape has been moving so quickly lately that by the time you read this in print, it will likely be out of date. If you're unfamiliar with some of the libraries mentioned here, that's fine.

At the time of PyTorch's first beta release:

- Theano and TensorFlow were the premiere low-level libraries, working with a model that had the user define a computational graph and then execute it.
- Lasagne and Keras were high-level wrappers around Theano, with Keras wrapping TensorFlow and CNTK as well.
- Caffe, Chainer, DyNet, Torch (the Lua-based precursor to PyTorch), MXNet, CNTK, DL4J, and others filled various niches in the ecosystem.

In the roughly two years that followed, the landscape changed drastically. The community largely consolidated behind either PyTorch or TensorFlow, with the adoption of other libraries dwindling, except for those filling specific niches. In a nutshell:

- Theano, one of the first deep learning frameworks, has ceased active development.
- TensorFlow:
  - Consumed Keras entirely, promoting it to a first-class API
  - Provided an immediate-execution “eager mode” that is somewhat similar to how PyTorch approaches computation
  - Released TF 2.0 with eager mode by default
- JAX, a library by Google that was developed independently from TensorFlow, has started gaining traction as a NumPy equivalent with GPU, autograd and JIT capabilities.
- PyTorch:
  - Consumed Caffe2 for its backend
  - Replaced most of the low-level code reused from the Lua-based Torch project
  - Added support for ONNX, a vendor-neutral model description and exchange format
  - Added a delayed-execution “graph mode” runtime called *TorchScript*
  - Released version 1.0
  - Replaced CNTK and Chainer as the framework of choice by their respective corporate sponsors

TensorFlow has a robust pipeline to production, an extensive industry-wide community, and massive mindshare. PyTorch has made huge inroads with the research and teaching communities, thanks to its ease of use, and has picked up momentum since, as researchers and graduates train students and move to industry. It has also built up steam in terms of production solutions. Interestingly, with the advent of TorchScript and eager mode, both PyTorch and TensorFlow have seen their feature sets start to converge with the other's, though the presentation of these features and the overall experience is still quite different between the two.

## 1.4 **An overview of how PyTorch supports deep learning projects**

We have already hinted at a few building blocks in PyTorch. Let’s now take some time to formalize a high-level map of the main components that form PyTorch. We can best do this by looking at what a deep learning project needs from PyTorch.

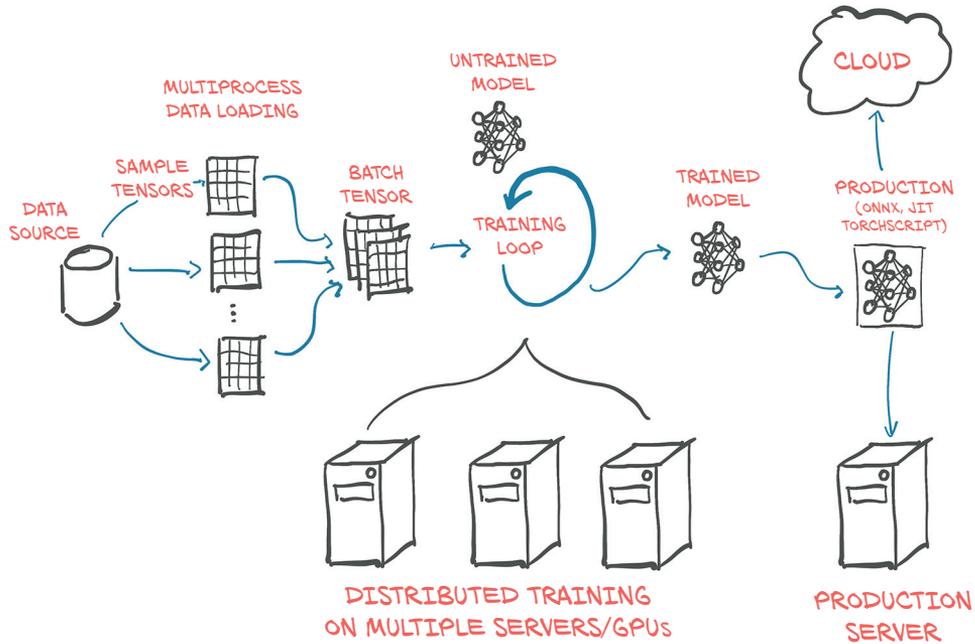
First, PyTorch has the “Py” as in Python, but there’s a lot of non-Python code in it. Actually, for performance reasons, most of PyTorch is written in C++ and CUDA ([www.geforce.com/hardware/technology/cuda](http://www.geforce.com/hardware/technology/cuda)), a C++-like language from NVIDIA that can be compiled to run with massive parallelism on GPUs. There are ways to run PyTorch directly from C++, and we’ll look into those in chapter 15. One of the motivations for this capability is to provide a reliable strategy for deploying models in production. However, most of the time we’ll interact with PyTorch from Python, building models, training them, and using the trained models to solve actual problems.

Indeed, the Python API is where PyTorch shines in term of usability and integration with the wider Python ecosystem. Let’s take a peek at the mental model of what PyTorch is.

As we already touched on, at its core, PyTorch is a library that provides *multidimensional arrays*, or *tensors* in PyTorch parlance (we’ll go into details on those in chapter 3), and an extensive library of operations on them, provided by the `torch` module. Both tensors and the operations on them can be used on the CPU or the GPU. Moving computations from the CPU to the GPU in PyTorch doesn’t require more than an additional function call or two. The second core thing that PyTorch provides is the ability of tensors to keep track of the operations performed on them and to analytically compute derivatives of an output of a computation with respect to any of its inputs. This is used for numerical optimization, and it is provided natively by tensors by virtue of dispatching through PyTorch’s *autograd* engine under the hood.

By having tensors and the autograd-enabled tensor standard library, PyTorch can be used for physics, rendering, optimization, simulation, modeling, and more—we’re very likely to see PyTorch used in creative ways throughout the spectrum of scientific applications. But PyTorch is first and foremost a deep learning library, and as such it provides all the building blocks needed to build neural networks and train them. Figure 1.2 shows a standard setup that loads data, trains a model, and then deploys that model to production.

The core PyTorch modules for building neural networks are located in `torch.nn`, which provides common neural network layers and other architectural components. Fully connected layers, convolutional layers, activation functions, and loss functions can all be found here (we’ll go into more detail about what all that means as we go through the rest of this book). These components can be used to build and initialize the untrained model we see in the center of figure 1.2. In order to train our model, we need a few additional things: a source of training data, an optimizer to adapt the model to the training data, and a way to get the model and data to the hardware that will actually be performing the calculations needed for training the model.



**Figure 1.2** Basic, high-level structure of a PyTorch project, with data loading, training, and deployment to production

At left in figure 1.2, we see that quite a bit of data processing is needed before the training data even reaches our model.<sup>4</sup> First we need to physically get the data, most often from some sort of storage as the data source. Then we need to convert each sample from our data into a something PyTorch can actually handle: tensors. This bridge between our custom data (in whatever format it might be) and a standardized PyTorch tensor is the `Dataset` class PyTorch provides in `torch.utils.data`. As this process is wildly different from one problem to the next, we will have to implement this data sourcing ourselves. We will look in detail at how to represent various type of data we might want to work with as tensors in chapter 4.

As data storage is often slow, in particular due to access latency, we want to parallelize data loading. But as the many things Python is well loved for do not include easy, efficient, parallel processing, we will need multiple processes to load our data, in order to assemble them into *batches*: tensors that encompass several samples. This is rather elaborate; but as it is also relatively generic, PyTorch readily provides all that magic in the `DataLoader` class. Its instances can spawn child processes to load data from a dataset in the background so that it's ready and waiting for the training loop as soon as the loop can use it. We will meet and use `Dataset` and `DataLoader` in chapter 7.

<sup>4</sup> And that's just the data preparation that is done on the fly, not the preprocessing, which can be a pretty large part in practical projects.

With the mechanism for getting batches of samples in place, we can turn to the training loop itself at the center of figure 1.2. Typically, the training loop is implemented as a standard Python `for` loop. In the simplest case, the model runs the required calculations on the local CPU or a single GPU, and once the training loop has the data, computation can start immediately. Chances are this will be your basic setup, too, and it's the one we'll assume in this book.

At each step in the training loop, we evaluate our model on the samples we got from the data loader. We then compare the outputs of our model to the desired output (the targets) using some *criterion* or *loss function*. Just as it offers the components from which to build our model, PyTorch also has a variety of loss functions at our disposal. They, too, are provided in `torch.nn`. After we have compared our actual outputs to the ideal with the loss functions, we need to push the model a little to move its outputs to better resemble the target. As mentioned earlier, this is where the PyTorch autograd engine comes in; but we also need an *optimizer* doing the updates, and that is what PyTorch offers us in `torch.optim`. We will start looking at training loops with loss functions and optimizers in chapter 5 and then hone our skills in chapters 6 through 8 before embarking on our big project in part 2.

It's increasingly common to use more elaborate hardware like multiple GPUs or multiple machines that contribute their resources to training a large model, as seen in the bottom center of figure 1.2. In those cases, `torch.nn.parallel.DistributedDataParallel` and the `torch.distributed` submodule can be employed to use the additional hardware.

The training loop might be the most unexciting yet most time-consuming part of a deep learning project. At the end of it, we are rewarded with a model whose parameters have been optimized on our task: the *trained model* depicted to the right of the training loop in the figure. Having a model to solve a task is great, but in order for it to be useful, we must put it where the work is needed. This *deployment* part of the process, depicted on the right in figure 1.2, may involve putting the model on a server or exporting it to load it to a cloud engine, as shown in the figure. Or we might integrate it with a larger application, or run it on a phone.

One particular step of the deployment exercise can be to export the model. As mentioned earlier, PyTorch defaults to an immediate execution model (eager mode). Whenever an instruction involving PyTorch is executed by the Python interpreter, the corresponding operation is immediately carried out by the underlying C++ or CUDA implementation. As more instructions operate on tensors, more operations are executed by the backend implementation.

PyTorch also provides a way to compile models ahead of time through *TorchScript*. Using TorchScript, PyTorch can serialize a model into a set of instructions that can be invoked independently from Python: say, from C++ programs or on mobile devices. We can think about it as a virtual machine with a limited instruction set, specific to tensor operations. This allows us to export our model, either as TorchScript to be used with the PyTorch runtime, or in a standardized format called *ONNX*. These features are at

the basis of the production deployment capabilities of PyTorch. We'll cover this in chapter 15.

## 1.5 Hardware and software requirements

This book will require coding and running tasks that involve heavy numerical computing, such as multiplication of large numbers of matrices. As it turns out, running a pretrained network on new data is within the capabilities of any recent laptop or personal computer. Even taking a pretrained network and retraining a small portion of it to specialize it on a new dataset doesn't necessarily require specialized hardware. You can follow along with everything we do in part 1 of this book using a standard personal computer or laptop.

However, we anticipate that completing a full training run for the more advanced examples in part 2 will require a CUDA-capable GPU. The default parameters used in part 2 assume a GPU with 8 GB of RAM (we suggest an NVIDIA GTX 1070 or better), but those can be adjusted if your hardware has less RAM available. To be clear: such hardware is not mandatory if you're willing to wait, but running on a GPU cuts training time by at least an order of magnitude (and usually it's 40–50x faster). Taken individually, the operations required to compute parameter updates are fast (from fractions of a second to a few seconds) on modern hardware like a typical laptop CPU. The issue is that training involves running these operations over and over, many, many times, incrementally updating the network parameters to minimize the training error.

Moderately large networks can take hours to days to train from scratch on large, real-world datasets on workstations equipped with a good GPU. That time can be reduced by using multiple GPUs on the same machine, and even further on clusters of machines equipped with multiple GPUs. These setups are less prohibitive to access than it sounds, thanks to the offerings of cloud computing providers. DAWNBench (<https://dawn.cs.stanford.edu/benchmark/index.html>) is an interesting initiative from Stanford University aimed at providing benchmarks on training time and cloud computing costs related to common deep learning tasks on publicly available datasets.

So, if there's a GPU around by the time you reach part 2, then great. Otherwise, we suggest checking out the offerings from the various cloud platforms, many of which offer GPU-enabled Jupyter Notebooks with PyTorch preinstalled, often with a free quota. Google Colaboratory (<https://colab.research.google.com>) is a great place to start.

The last consideration is the operating system (OS). PyTorch has supported Linux and macOS from its first release, and it gained Windows support in 2018. Since current Apple laptops do not include GPUs that support CUDA, the precompiled macOS packages for PyTorch are CPU-only. Throughout the book, we will try to avoid assuming you are running a particular OS, although some of the scripts in part 2 are shown as if running from a Bash prompt under Linux. Those scripts' command lines should convert to a Windows-compatible form readily. For convenience, code will be listed as if running from a Jupyter Notebook when possible.

For installation information, please see the Get Started guide on the official PyTorch website (<https://pytorch.org/get-started/locally>). We suggest that Windows users install with Anaconda or Miniconda (<https://www.anaconda.com/distribution> or <https://docs.conda.io/en/latest/miniconda.html>). Other operating systems like Linux typically have a wider variety of workable options, with Pip being the most common package manager for Python. We provide a requirements.txt file that pip can use to install dependencies. Of course, experienced users are free to install packages in the way that is most compatible with your preferred development environment.

Part 2 has some nontrivial download bandwidth and disk space requirements as well. The raw data needed for the cancer-detection project in part 2 is about 60 GB to download, and when uncompressed it requires about 120 GB of space. The compressed data can be removed after decompressing it. In addition, due to caching some of the data for performance reasons, another 80 GB will be needed while training. You will need a total of 200 GB (at minimum) of free disk space on the system that will be used for training. While it is possible to use network storage for this, there might be training speed penalties if the network access is slower than local disk. Preferably you will have space on a local SSD to store the data for fast retrieval.

### 1.5.1 *Using Jupyter Notebooks*

We're going to assume you've installed PyTorch and the other dependencies and have verified that things are working. Earlier we touched on the possibilities for following along with the code in the book. We are going to be making heavy use of Jupyter Notebooks for our example code. A Jupyter Notebook shows itself as a page in the browser through which we can run code interactively. The code is evaluated by a *kernel*, a process running on a server that is ready to receive code to execute and send back the results, which are then rendered inline on the page. A notebook maintains the state of the kernel, like variables defined during the evaluation of code, in memory until it is terminated or restarted. The fundamental unit with which we interact with a notebook is a *cell*: a box on the page where we can type code and have the kernel evaluate it (through the menu item or by pressing Shift-Enter). We can add multiple cells in a notebook, and the new cells will see the variables we created in the earlier cells. The value returned by the last line of a cell will be printed right below the cell after execution, and the same goes for plots. By mixing source code, results of evaluations, and Markdown-formatted text cells, we can generate beautiful interactive documents. You can read everything about Jupyter Notebooks on the project website (<https://jupyter.org>).

At this point, you need to start the notebook server from the root directory of the code checkout from GitHub. How exactly starting the server looks depends on the details of your OS and how and where you installed Jupyter. If you have questions, feel free to ask on the book's forum.<sup>5</sup> Once started, your default browser will pop up, showing a list of local notebook files.

---

<sup>5</sup> <https://forums.manning.com/forums/deep-learning-with-pytorch>

**NOTE** Jupyter Notebooks are a powerful tool for expressing and investigating ideas through code. While we think that they make for a good fit for our use case with this book, they're not for everyone. We would argue that it's important to focus on removing friction and minimizing cognitive overhead, and that's going to be different for everyone. Use what you like during your experimentation with PyTorch.

Full working code for all listings from the book can be found at the book's website ([www.manning.com/books/deep-learning-with-pytorch](http://www.manning.com/books/deep-learning-with-pytorch)) and in our repository on GitHub (<https://github.com/deep-learning-with-pytorch/dlwpt-code>).

## 1.6 Exercises

- 1 Start Python to get an interactive prompt.
  - a What Python version are you using? We hope it is at least 3.6!
  - b Can you `import torch`? What version of PyTorch do you get?
  - c What is the result of `torch.cuda.is_available()`? Does it match your expectation based on the hardware you're using?
- 2 Start the Jupyter notebook server.
  - a What version of Python is Jupyter using?
  - b Is the location of the `torch` library used by Jupyter the same as the one you imported from the interactive prompt?

## 1.7 Summary

- Deep learning models automatically learn to associate inputs and desired outputs from examples.
- Libraries like PyTorch allow you to build and train neural network models efficiently.
- PyTorch minimizes cognitive overhead while focusing on flexibility and speed. It also defaults to immediate execution for operations.
- TorchScript allows us to precompile models and invoke them not only from Python but also from C++ programs and on mobile devices.
- Since the release of PyTorch in early 2017, the deep learning tooling ecosystem has consolidated significantly.
- PyTorch provides a number of utility libraries to facilitate deep learning projects.

# Pretrained networks

---

## ***This chapter covers***

- Running pretrained image-recognition models
- An introduction to GANs and CycleGAN
- Captioning models that can produce text descriptions of images
- Sharing models through Torch Hub

We closed our first chapter promising to unveil amazing things in this chapter, and now it's time to deliver. Computer vision is certainly one of the fields that have been most impacted by the advent of deep learning, for a variety of reasons. The need to classify or interpret the content of natural images existed, very large datasets became available, and new constructs such as convolutional layers were invented and could be run quickly on GPUs with unprecedented accuracy. All of these factors combined with the internet giants' desire to understand pictures taken by millions of users with their mobile devices and managed on said giants' platforms. Quite the perfect storm.

We are going to learn how to use the work of the best researchers in the field by downloading and running very interesting models that have already been trained on open, large-scale datasets. We can think of a pretrained neural network as similar to

a program that takes inputs and generates outputs. The behavior of such a program is dictated by the architecture of the neural network and by the examples it saw during training, in terms of desired input-output pairs, or desired properties that the output should satisfy. Using an off-the-shelf model can be a quick way to jump-start a deep learning project, since it draws on expertise from the researchers who designed the model, as well as the computation time that went into training the weights.

In this chapter, we will explore three popular pretrained models: a model that can label an image according to its content, another that can fabricate a new image from a real image, and a model that can describe the content of an image using proper English sentences. We will learn how to load and run these pretrained models in PyTorch, and we will introduce PyTorch Hub, a set of tools through which PyTorch models like the pretrained ones we'll discuss can be easily made available through a uniform interface. Along the way, we'll discuss data sources, define terminology like *label*, and attend a zebra rodeo.

If you're coming to PyTorch from another deep learning framework, and you'd rather jump right into learning the nuts and bolts of PyTorch, you can get away with skipping to the next chapter. The things we'll cover in this chapter are more fun than foundational and are somewhat independent of any given deep learning tool. That's not to say they're not important! But if you've worked with pretrained models in other deep learning frameworks, then you already know how powerful a tool they can be. And if you're already familiar with the generative adversarial network (GAN) game, you don't need us to explain it to you.

We hope you keep reading, though, since this chapter hides some important skills under the fun. Learning how to run a pretrained model using PyTorch is a useful skill—full stop. It's especially useful if the model has been trained on a large dataset. We will need to get accustomed to the mechanics of obtaining and running a neural network on real-world data, and then visualizing and evaluating its outputs, whether we trained it or not.

## 2.1 A pretrained network that recognizes the subject of an image

As our first foray into deep learning, we'll run a state-of-the-art deep neural network that was pretrained on an object-recognition task. There are many pretrained networks that can be accessed through source code repositories. It is common for researchers to publish their source code along with their papers, and often the code comes with weights that were obtained by training a model on a reference dataset. Using one of these models could enable us to, for example, equip our next web service with image-recognition capabilities with very little effort.

The pretrained network we'll explore here was trained on a subset of the ImageNet dataset (<http://imagenet.stanford.edu>). ImageNet is a very large dataset of over 14 million images maintained by Stanford University. All of the images are labeled with a hierarchy of nouns that come from the WordNet dataset (<http://wordnet.princeton.edu>), which is in turn a large lexical database of the English language.

The ImageNet dataset, like several other public datasets, has its origin in academic competitions. Competitions have traditionally been some of the main playing fields where researchers at institutions and companies regularly challenge each other. Among others, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) has gained popularity since its inception in 2010. This particular competition is based on a few tasks, which can vary each year, such as image classification (telling what object categories the image contains), object localization (identifying objects' position in images), object detection (identifying and labeling objects in images), scene classification (classifying a situation in an image), and scene parsing (segmenting an image into regions associated with semantic categories, such as cow, house, cheese, hat). In particular, the image-classification task consists of taking an input image and producing a list of 5 labels out of 1,000 total categories, ranked by confidence, describing the content of the image.

The training set for ILSVRC consists of 1.2 million images labeled with one of 1,000 nouns (for example, “dog”), referred to as the *class* of the image. In this sense, we will use the terms *label* and *class* interchangeably. We can take a peek at images from ImageNet in figure 2.1.

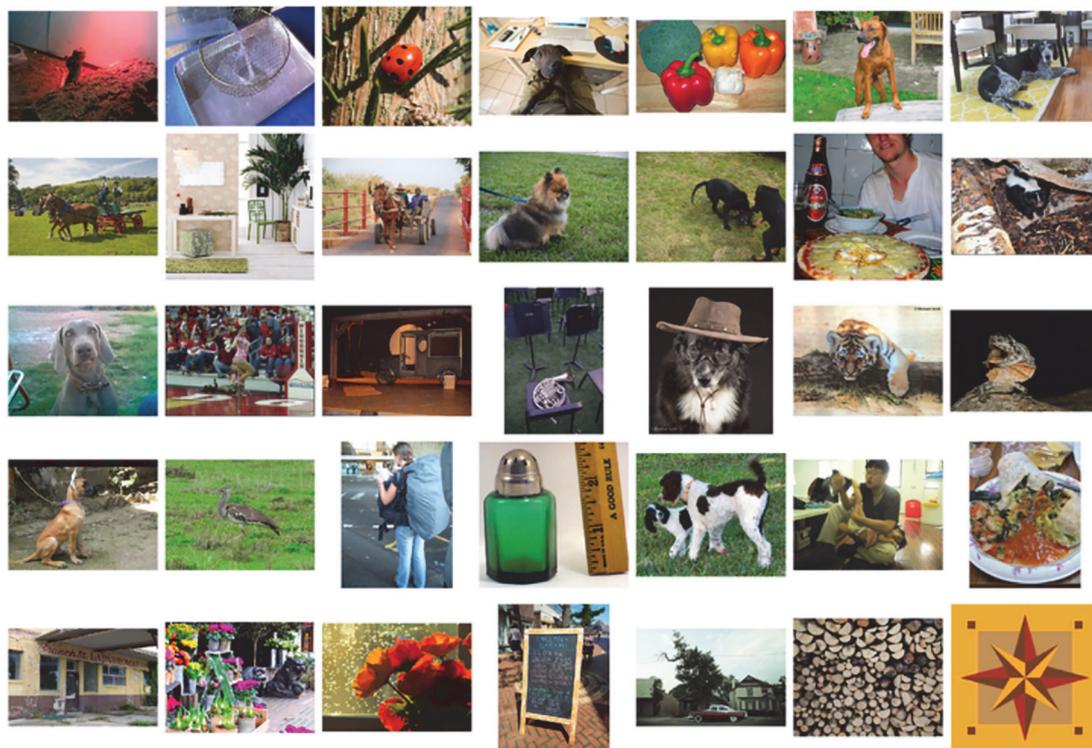


Figure 2.1 A small sample of ImageNet images

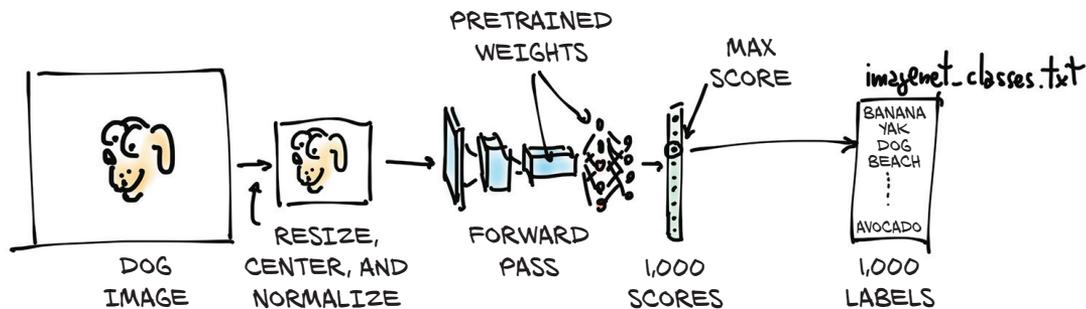


Figure 2.2 The inference process

We are going to end up being able to take our own images and feed them into our pretrained model, as pictured in figure 2.2. This will result in a list of predicted labels for that image, which we can then examine to see what the model thinks our image is. Some images will have predictions that are accurate, and others will not!

The input image will first be preprocessed into an instance of the multidimensional array class `torch.Tensor`. It is an RGB image with height and width, so this tensor will have three dimensions: the three color channels, and two spatial image dimensions of a specific size. (We'll get into the details of what a tensor is in chapter 3, but for now, think of it as being like a vector or matrix of floating-point numbers.) Our model will take that processed input image and pass it into the pretrained network to obtain scores for each class. The highest score corresponds to the most likely class according to the weights. Each class is then mapped one-to-one onto a class label. That output is contained in a `torch.Tensor` with 1,000 elements, each representing the score associated with that class.

Before we can do all that, we'll need to get the network itself, take a peek under the hood to see how it's structured, and learn about how to prepare our data before the model can use it.

### 2.1.1 Obtaining a pretrained network for image recognition

As discussed, we will now equip ourselves with a network trained on ImageNet. To do so, we'll take a look at the TorchVision project (<https://github.com/pytorch/vision>), which contains a few of the best-performing neural network architectures for computer vision, such as AlexNet (<http://mng.bz/lo6z>), ResNet (<https://arxiv.org/pdf/1512.03385.pdf>), and Inception v3 (<https://arxiv.org/pdf/1512.00567.pdf>). It also has easy access to datasets like ImageNet and other utilities for getting up to speed with computer vision applications in PyTorch. We'll dive into some of these further along in the book. For now, let's load up and run two networks: first AlexNet, one of the early breakthrough networks for image recognition; and then a residual network, ResNet for short, which won the ImageNet classification, detection, and localization

competitions, among others, in 2015. If you didn't get PyTorch up and running in chapter 1, now is a good time to do that.

The predefined models can be found in `torchvision.models` (code/p1ch2/2\_pre\_trained\_networks.ipynb):

```
# In[1]:
from torchvision import models
```

We can take a look at the actual models:

```
# In[2]:
dir(models)

# Out[2]:
['AlexNet',
 'DenseNet',
 'Inception3',
 'ResNet',
 'SqueezeNet',
 'VGG',
 ...
 'alexnet',
 'densenet',
 'densenet121',
 ...
 'resnet',
 'resnet101',
 'resnet152',
 ...
]
```

The capitalized names refer to Python classes that implement a number of popular models. They differ in their architecture—that is, in the arrangement of the operations occurring between the input and the output. The lowercase names are convenience functions that return models instantiated from those classes, sometimes with different parameter sets. For instance, `resnet101` returns an instance of `ResNet` with 101 layers, `resnet18` has 18 layers, and so on. We'll now turn our attention to `AlexNet`.

### 2.1.2 *AlexNet*

The `AlexNet` architecture won the 2012 ILSVRC by a large margin, with a top-5 test error rate (that is, the correct label must be in the top 5 predictions) of 15.4%. By comparison, the second-best submission, which wasn't based on a deep network, trailed at 26.2%. This was a defining moment in the history of computer vision: the moment when the community started to realize the potential of deep learning for vision tasks. That leap was followed by constant improvement, with more modern architectures and training methods getting top-5 error rates as low as 3%.