

PyTorch Computer Vision Cookbook

Over 70 recipes to master the art of computer vision with deep learning and PyTorch 1.x



Packt

www.packt.com

Michael Avendi

PyTorch Computer Vision Cookbook

Over 70 recipes to master the art of computer vision with
deep learning and PyTorch 1.x

Michael Avendi

Packt

BIRMINGHAM - MUMBAI

PyTorch Computer Vision Cookbook

Copyright © 2020 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Amey Varangaonkar

Acquisition Editor: Devika Battike

Content Development Editor: Nathanya Dias

Senior Editor: Ayaan Hoda

Technical Editor: Manikandan Kurup

Copy Editor: Safis Editing

Project Coordinator: Aishwarya Mohan

Proofreader: Safis Editing

Indexer: Manju Arasan

Production Designer: Jyoti Chauhan

First published: March 2020

Production reference: 1200320

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-83864-483-3

www.packtpub.com



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

About the author

Michael Avendi is a principal data scientist with vast experience in deep learning, computer vision, and medical imaging analysis. He works on the research and development of data-driven algorithms for various imaging problems, including medical imaging applications. His research papers have been published in major medical journals, including the *Medical Imaging Analysis* journal. Michael Avendi is an active Kaggle participant and was awarded a top prize in a Kaggle competition in 2017.

About the reviewers

Amin Ahmadi Tazehkandi is an Iranian author, software engineer and computer vision expert. He has worked in numerous software companies across the globe and has a long list of awards and achievements, including a countrywide hackathon win, and an award-winning paper as well. Amin is an avid blogger and long-time contributor to the open-source, cross-platform and computer vision developer communities. He is the proud author of *Computer Vision with OpenCV 3 and Qt5*, and *Hands-On Algorithms for Computer Vision*.

I would like to thank the love of my life and partner, Senem, for her patience and support. I would also like to thank little Naz, for helping her daddy type and work faster than ever.

Dr. Matthew Rever has a Ph.D. in Electrical Engineering from the University of Michigan, Ann Arbor. His career revolves around image processing, computer vision, and machine learning for scientific research purposes. He is well versed in C++, a language he has used over the last 20 years. Over the past few years, he has been using Matlab, Python, OpenCV, SciPy, scikit-learn, TensorFlow, and PyTorch. He believes it is important to stay with keep updated with the latest tools available in order to be most productive. He is also the author of Packt's *Computer Vision Projects with Python 3* and *Advanced Computer Vision Projects*.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
Chapter 1: Getting Started with PyTorch for Deep Learning	8
Technical requirements	9
Installing software tools and packages	10
How to do it...	10
Installing Anaconda	10
Installing PyTorch	11
Verifying the installation	12
Installing other packages	13
How it works...	13
Working with PyTorch tensors	14
How to do it...	15
Defining the tensor data type	15
Changing the tensor's data type	15
Converting tensors into NumPy arrays	16
Converting NumPy arrays into tensors	16
Moving tensors between devices	17
How it works...	18
See also	18
Loading and processing data	19
How to do it...	19
Loading a dataset	20
Data transformation	22
Wrapping tensors into a dataset	24
Creating data loaders	24
How it works...	25
Building models	26
How to do it...	26
Defining a linear layer	26
Defining models using nn.Sequential	27
Defining models using nn.Module	28
Moving the model to a CUDA device	29
Printing the model summary	30
How it works...	31
Defining the loss function and optimizer	32
How to do it...	32
Defining the loss function	32
Defining the optimizer	33
How it works...	33
See also	34
Training and evaluation	34

How to do it...	34
Storing and loading models	36
Deploying the model	37
How it works...	38
There's more...	39
Chapter 2: Binary Image Classification	
Exploring the dataset	41
Getting ready	42
How to do it...	42
How it works...	45
Creating a custom dataset	46
How to do it...	46
How it works...	48
Splitting the dataset	49
How to do it...	49
How it works...	52
Transforming the data	52
How to do it...	52
How it works...	53
Creating dataloaders	53
How to do it...	54
How it works...	54
Building the classification model	55
How to do it...	56
How it works...	60
See also	62
Defining the loss function	62
How to do it...	63
How it works...	64
See also	64
Defining the optimizer	65
How to do it...	65
How it works...	66
See also	66
Training and evaluation of the model	67
How to do it...	67
How it works...	74
There's more...	76
Deploying the model	76
How to do it...	76
How it works...	79
Model inference on test data	79
Getting ready	80
How to do it...	80

How it works...	81
See also	82
Chapter 3: Multi-Class Image Classification	84
Loading and processing data	85
How to do it...	85
How it works...	92
There's more...	93
See also	93
Building the model	94
How to do it...	94
How it works...	99
There's more...	100
See also	100
Defining the loss function	100
How to do it...	101
How it works...	102
See also	102
Defining the optimizer	103
How to do it...	103
How it works...	104
See also	105
Training and transfer learning	105
How to do it...	106
How it works...	112
See also	114
Deploying the model	114
How to do it...	114
How it works...	118
Chapter 4: Single-Object Detection	120
Exploratory data analysis	122
Getting ready	122
How to do it...	123
How it works...	129
Data transformation for object detection	130
How to do it...	130
How it works...	136
There's more...	137
See also	140
Creating custom datasets	141
How to do it...	141
How it works...	147
Creating the model	149
How to do it...	149
How it works...	152

Defining the loss, optimizer, and IOU metric	153
How to do it...	154
How it works...	157
Training and evaluation of the model	158
How to do it...	158
How it works...	163
Deploying the model	164
How to do it...	164
How it works...	169
Chapter 5: Multi-Object Detection	170
 Creating datasets	171
Getting ready	171
How to do it...	173
Creating a custom COCO dataset	173
Transforming data	180
Defining the Dataloaders	184
How it works...	185
 Creating a YOLO-v3 model	189
How to do it...	190
Parsing the configuration file	190
Creating PyTorch modules	191
Defining the Darknet model	192
How it works...	193
 Defining the loss function	199
How to do it...	200
How it works...	204
 Training the model	208
How to do it...	208
How it works...	211
 Deploying the model	213
How to do it...	213
How it works...	218
See also	220
Chapter 6: Single-Object Segmentation	221
 Creating custom datasets	222
Getting ready	222
How to do it...	223
Data exploration	223
Data augmentation	225
Creating the datasets	226
How it works...	230
 Defining the model	232
How to do it...	233
How it works...	236

Defining the loss function and optimizer	237
How to do it...	237
How it works...	239
Training the model	240
How to do it...	241
How it works...	245
Deploying the model	245
How to do it...	246
How it works...	248
Chapter 7: Multi-Object Segmentation	249
 Creating custom datasets	250
How to do it...	251
How it works...	256
 Defining and deploying a model	258
How to do it...	259
How it works...	261
See also	262
 Defining the loss function and optimizer	262
How to do it...	263
How it works...	264
 Training the model	265
How to do it...	265
How it works...	268
Chapter 8: Neural Style Transfer with PyTorch	270
 Loading the data	271
Getting ready	271
How to do it...	271
How it works...	275
 Implementing neural style transfer	276
How to do it...	277
Loading the pretrained model	277
Defining loss functions	278
Defining the optimizer	280
Running the algorithm	280
How it works...	282
See also	285
Chapter 9: GANs and Adversarial Examples	286
 Creating the dataset	287
How to do it...	288
How it works...	290
 Defining the generator and discriminator	291
How to do it...	291
How it works...	295

Defining the loss and optimizer	296
How to do it...	296
How it works...	296
Training the models	297
How to do it...	297
How it works...	300
See also	301
Deploying the generator	301
How to do it...	301
How it works...	303
Attacking models with adversarial examples	303
Getting ready	304
How to do it...	305
Loading the dataset	305
Loading the pre-trained model	305
Implementing the attack	307
How it works...	309
There's more...	311
Chapter 10: Video Processing with PyTorch	312
Creating the dataset	313
Getting ready	314
How to do it...	315
Preparing the data	315
Splitting the data	317
Defining the PyTorch datasets	319
Defining the data loaders	324
How it works...	326
Defining the model	328
How to do it...	329
How it works...	332
Training the model	332
How to do it...	333
How it works...	334
Deploying the video classification model	334
How to do it...	334
How it works...	336
Other Books You May Enjoy	337
Index	340

Preface

This book will enable you to solve the trickiest of problems in **computer vision** (CV) using deep learning algorithms and techniques. You will learn how to use several different algorithms for different computer vision problems such as classification, detection, segmentation, and others using PyTorch. The book is packed with best practices for the training and deployment of computer vision applications.

Starting with a quick overview of the PyTorch library and key deep learning concepts, the book covers common and not-so-common challenges faced while performing image recognition, image segmentation, captioning, image generation, and many other tasks. You'll implement these tasks using various deep learning architectures such as **convolutional neural networks** (CNNs), **recurrent neural networks** (RNNs), **long short-term memory** (LSTM), and **generative adversarial networks** (GANs). Using a problem-solution approach, you'll solve any issue you might face while fine-tuning the performance of a model or integrating a model into your application.

Additionally, you'll even get to grips with scaling a model to handle larger workloads and implement best practices for training models efficiently.

Who this book is for

Computer vision professionals, data scientists, deep learning engineers, and AI developers looking for quick solutions for various computer vision problems will find this book useful. Intermediate knowledge of computer vision concepts along with Python programming experience is required.

What this book covers

Chapter 1, *Getting Started with PyTorch for Deep Learning*, introduces the PyTorch library and gives a detailed description of the journey of installing and setting up a deep learning environment with it. It also introduces some of the key components of the library that will help you on the rest of your learning journey.

Chapter 2, *Binary Image Classification*, explains how to implement a binary image classification network using PyTorch. You will learn how to load and preprocess datasets and implement classification models based on CNNs. You will also implement an objective function for binary classification. Moreover, you will set hyperparameters such as learning rates, batch size, epochs, and the number of filters and layers. Then, you will train and validate the model and store it. Finally, you will deploy the model on new data.

Chapter 3, *Multi-Class Image Classification*, walks you through implementing a multi-class image classification network using PyTorch. You will learn how to load and preprocess datasets and implement classification models based on CNNs. You will also implement an objective function for multi-class classification. Moreover, you will set hyperparameters such as learning rates, batch size, epochs, and the number of filters and layers. Then, you will train and validate the model and store it. Finally, you will deploy the model on new data.

Chapter 4, *Single-Object Detection*, covers implementing a single-object detection network using PyTorch. You will learn how to load and preprocess datasets and implement object detection models based on CNNs. You will also implement an objective function for single-object detection. Moreover, you will set hyperparameters such as learning rates, batch size, epochs, and the number of filters and layers. Then, you will train and validate the model and store it. Finally, you will deploy the model on new data.

Chapter 5, *Multi-Object Detection*, guides you through implementing a multi-object detection network using PyTorch. You will learn how to load and preprocess datasets and implement object detection models based on CNNs and YOLO networks. You will also implement an objective function for multi-object detection. Moreover, you will set hyperparameters such as learning rates, batch size, epochs, and the number of filters and layers. Then, you will train and validate the model and store it. Finally, you will deploy the model on new data.

Chapter 6, *Single-Object Segmentation*, explains implementing a single-object segmentation network using PyTorch. You will learn how to load and preprocess datasets, and implement object segmentation models based on CNNs and UNets. You will also implement an objective function for single-object segmentation. Moreover, you will set hyperparameters such as learning rates, batch size, epochs, and the number of filters and layers. Then, you will train and validate the model and store it. Finally, you will deploy the model on new data.

Chapter 7, *Multi-Object Segmentation*, covers implementing a multi-object segmentation network using PyTorch. You will learn how to load and preprocess datasets and implement object segmentation models based on CNNs and UNets. You will also implement an objective function for multi-object segmentation. Moreover, you will set hyperparameters such as learning rates, batch size, epochs, and the number of filters and layers. Then, you will train and validate the model and store it. Finally, you will deploy the model on new data.

Chapter 8, *Neural Style Transfer with PyTorch*, explains how to implement a neural style transfer network using PyTorch. You will learn how to load and preprocess datasets and implement neural style transfer models based on CNN and pre-trained models. You will also implement the content and style objective functions of neural style transfer. Moreover, you will set hyperparameters such as learning rates, batch size, epochs, and the number of filters and layers. Then, you will train the model and store it. Finally, you will deploy the model to style new images.

Chapter 9, *GANs and Adversarial Examples*, covers implementing a GAN using PyTorch. You will learn how to load and preprocess datasets and implement the discriminator and generator models of a GAN based on CNN. You will also implement the objective functions of a GAN. Moreover, you will set hyperparameters such as learning rates, batch size, epochs, and the number of filters and layers. Then, you will train the model and store it. Finally, you will deploy the model to generate new images. In addition, you will implement adversarial examples to fool deep learning models in Pytorch. Specifically, you will implement a white-box attack with the goal of misclassification referred to as the **Fast Gradient Sign Attack (FGSM)**. You will also implement the method to add imperceptible perturbations to an image and cause drastically different model performance.

Chapter 10, *Video Processing with PyTorch*, explains how to implement a video recognition network using PyTorch. You will learn how to load and preprocess datasets and implement video classification models based on 3D-CNN and RNN/LSTM. You will also implement an objective function for video classification. Moreover, you will set hyperparameters such as learning rates, batch size, epochs, and the number of filters and layers. Then, you will train and validate the model and store it. Finally, you will deploy the model on new data.

To get the most out of this book

You will need a basic understanding of computer vision and working knowledge of Python programming. Having a basic understanding of concepts such as object detection and image generation would be an added advantage. All of the code works with the latest version of PyTorch, that is, PyTorch 1.x.

Software/hardware covered in the book	OS requirements
PyTorch	Any
Python	Any
Minimum 8 GB system (GPU preferred)	

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying/pasting of code.

Computer vision and Python knowledge is a must.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/ManikandanKurup-Packt/PyTorch-Computer-Vision-Cookbook>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781838644833_ColorImages.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The `train_labels.csv` file provides the ground truth for the images in the `train` folder"

A block of code is set as follows:

```
import pandas as pd

path2csv="../data/train_labels.csv"
labels_df=pd.read_csv(path2csv)
labels_df.head()
```

Any command-line input or output is written as follows:

```
$ conda install scikit-learn
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Choose **Late Submission** from the menu and upload the file."

Warnings or important notes appear like this.





Tips and tricks appear like this.

Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

Getting Started with PyTorch for Deep Learning

There has been significant progress in computer vision because of deep learning in recent years. This helped to improve the performance of various tasks such as image recognition, object detection, image segmentation, and image generation. Deep learning frameworks and libraries have played a major role in this process. PyTorch, as a deep learning library, has emerged since 2016 and gained great attention among deep learning practitioners due to its flexibility and ease of use.

There are several frameworks that practitioners use to build deep learning algorithms. In this book, we will use the latest version of PyTorch 1.0 to develop and train various deep learning models. PyTorch is a deep learning framework developed by Facebook's artificial intelligence research group. It provides flexibility and ease of use at the same. If you are familiar with other deep learning frameworks, you will find PyTorch very enjoyable.

In this chapter, we will provide a review of deep learning concepts and their implementation using PyTorch 1.0. We will cover the following recipes:

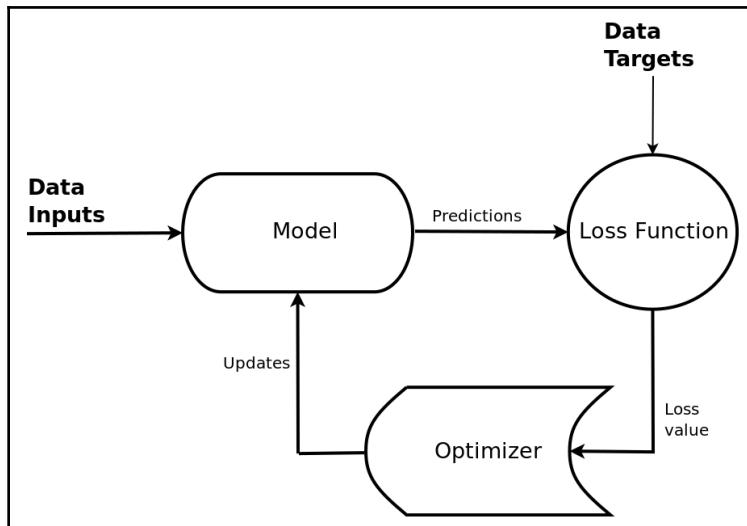
- Installing software tools and packages
- Working with PyTorch tensors
- Loading and processing data
- Building models
- Defining the loss function and optimizer
- Training and evaluation

Developing deep learning algorithms is comprised of two steps: training and deployment. In the training step, we use training data to train a model or network. In the deployment step, we deploy the trained model to predict the target values for new inputs.

To train deep learning algorithms, the following ingredients are required:

- Training data (inputs and targets)
- The model (also called the network)
- The loss function (also called the objective function or criterion)
- The optimizer

You can see the interaction between these elements in the following diagram:



The training process for deep learning algorithms is an iterative process. In each iteration, we select a batch of training data. Then, we feed the data to the model to get the model output. After that, we calculate the loss value. Next, we compute the gradients of the loss function with respect to the model parameters (also known as the weights). Finally, the optimizer updates the parameters based on the gradients. This loop continues. We also use a validation dataset to track the model's performance during training. We stop the training process when the performance plateaus.

Technical requirements

It is assumed that you are familiar with deep learning and computer vision concepts. In addition, you are expected to have medium proficiency in Python programming.

Deep learning algorithms are computationally heavy. You will need a computer with decent GPU hardware to build deep learning algorithms in a reasonable time. The training time depends on the model and data size. We recommend equipping your computer with an NVIDIA GPU or using services such as AWS to rent a computer on the cloud. Also, make sure to install the NVIDIA driver and CUDA. For the rest of this book, whenever referring to GPU/CUDA, we assume that you have installed the required drivers. You can still use your computer with a CPU. However, the training time will be much longer and you'll need to be patient!

Installing software tools and packages

We will use Python, PyTorch, and other Python packages to develop various deep learning algorithms in this book. Therefore, first, we need to install several software tools, including Anaconda, PyTorch, and Jupyter Notebook, before conducting any deep learning implementation.

How to do it...

In the following sections, we will provide instructions on how to install the required software tools.

Installing Anaconda

Let's look at the following steps to install Anaconda:

1. To install Anaconda, visit the following link: <https://www.anaconda.com/distribution/>.
2. In the link provided, you will find three distributions: Windows, macOS, and Linux. Select the desired distribution. You can download either Python 2.x or Python 3.x. We recommend using a Linux system and downloading and installing Python 3.x version.
3. After installing Anaconda, create a conda environment for PyTorch experiments. For instance, in the following code block, we create `conda-pytorch` as follows:

```
# choose your desired python version  
$ conda create env conda-pytorch python=3.6
```

4. Activate the environment on Linux/Mac using the following command:

```
# activate conda environment on Linux/mac  
$ source activate conda-pytorch  
# After activation, (conda-pytorch) will be added to the prompt.
```

5. Activate the environment on Windows using the following command:

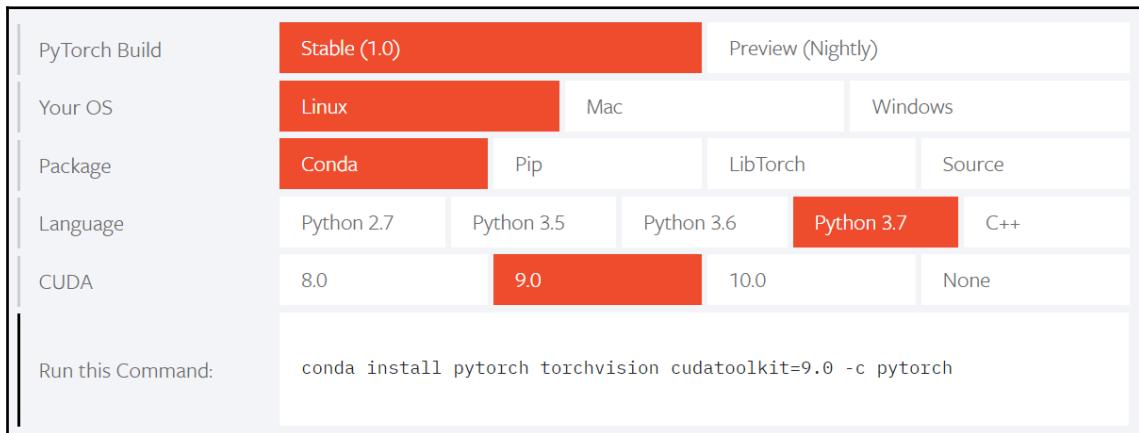
```
# activate conda environment on Windows  
$ activate conda-pytorch  
# After activation, (conda-pytorch) will be added to the prompt.
```

In the next section, we will show you how to install PyTorch.

Installing PyTorch

Now, let's look at the installation of PyTorch:

1. To install PyTorch, click on the following link: <https://pytorch.org/>.
2. Scroll down to the **Quick Start Locally** section. From the interactive table, select the options that are appropriate for your computer system.
3. For instance, if we would like to install PyTorch 1.0 on a Linux OS using the Conda package, along with Python 3.7 and CUDA 9.0, we can make the selections shown in the following screenshot:



4. Copy the given command at the end of the table and run it in your Terminal:

```
$ conda install pytorch torchvision cudatoolkit=9.0 -c pytorch
```

5. This will install PyTorch 1.0 and torchvision.

In the next section, we will show you how to verify the installation.

Verifying the installation

Let's make sure that the installation is correct by importing PyTorch into Python:

1. Launch Python from a Terminal:

```
# launch python from a terminal (linux/macos) or anaconda prompt  
(windows)  
$ python  
>>>
```

2. Import torch and get its version:

```
# import PyTorch  
>>> import torch  
  
# get PyTorch version  
>>> torch.__version__  
'1.0.1.post2'
```

3. Import torchvision and get its version:

```
# import torchvision  
>>> import torchvision  
  
# get torchvision version  
>>> torchvision.__version__  
'0.2.2'
```

4. Check if CUDA is available:

```
# checking if cuda is available  
>>> torch.cuda.is_available()
```

5. Get the number of CUDA devices:

```
# get number of cuda/gpu devices  
>>> torch.cuda.device_count()  
1
```

6. Get the CUDA device id:

```
# get cuda/gpu device id
>>> torch.cuda.current_device()
0
```

7. Get the CUDA device name:

```
# get cuda/gpu device name
>>> torch.cuda.get_device_name(0)
'GeForce GTX TITAN X'
```

In the next section, we will install other packages.

Installing other packages

The majority of packages are installed using Anaconda. However, we may have to manually install other packages as we continue in this book:

1. In this book, we will use Jupyter Notebook to implement our code. Install Jupyter Notebook in the environment:

```
# install Jupyter Notebook in the conda environment
$ conda install -c anaconda jupyter
```

2. Install matplotlib to show images and plots:

```
# install matplotlib
$ conda install -c conda-forge matplotlib
```

3. Install pandas to work with DataFrames:

```
# install pandas
$ conda install -c anaconda pandas
```

In the next section, we will explain each step in detail.

How it works...

We started with the installation of Anaconda by following the necessary steps. After installing Anaconda, we created a conda environment for PyTorch experiments. You can use any operating system such as Windows, macOS, or Linux; we recommended Linux for this book. Next, we installed PyTorch in the conda environment.

Next, we verified the installation of PyTorch. We launched Python from a Terminal or Anaconda Prompt. Then, we imported `torch` and `torchvision` and obtained the package versions. Next, we checked the availability and number of CUDA devices. Our system is equipped with one GPU. Also, we got the CUDA device `id` and name. Our GPU device is GeForce GTX TITAN X. The default device ID is zero.

Finally, we installed Jupyter Notebook, `matplotlib`, and `pandas` in the conda environment. We will be developing the scripts in this book in Jupyter Notebook. We will also be using `matplotlib` to plot graphs and show images, as well as `pandas` to work with DataFrames.

Working with PyTorch tensors

PyTorch is built on tensors. A PyTorch tensor is an n -dimensional array, similar to NumPy arrays.

If you are familiar with NumPy, you will see a similarity in the syntax when working with tensors, as shown in the following table:

NumPy Arrays	PyTorch tensors	Description
<code>numpy.ones(..)</code>	<code>torch.ones(..)</code>	Create an array of ones
<code>numpy.zeros(..)</code>	<code>torch.zeros(..)</code>	Create an array of zeros
<code>numpy.random.rand(..)</code>	<code>torch.rand(..)</code>	Create a random array
<code>numpy.array(..)</code>	<code>torch.tensor(..)</code>	Create an array from given values
<code>x.shape</code>	<code>x.shape or x.size()</code>	Get an array shape

In this recipe, you will learn how to define and change tensors, convert tensors into arrays, and move them between computing devices.

How to do it...

For the following code examples, you can either launch Python or the Jupyter Notebook app from a Terminal.

Defining the tensor data type

The default tensor data type is `torch.float32`. This is the most used data type for tensor operations. Let's take a look:

1. Define a tensor with a default data type:

```
x = torch.ones(2, 2)
print(x)
print(x.dtype)

tensor([[1., 1.],
        [1., 1.]])
torch.float32
```

2. Specify the data type when defining a tensor:

```
# define a tensor with specific data type
x = torch.ones(2, 2, dtype=torch.int8)
print(x)
print(x.dtype)

tensor([[1, 1],
        [1, 1]], dtype=torch.int8)
torch.int8
```

In the next section, we will show you how to change the tensor's type.

Changing the tensor's data type

We can change a tensor's data type using the `.type` method:

1. Define a tensor with the `torch.uint8` type:

```
x=torch.ones(1,dtype=torch.uint8)
print(x.dtype)
torch.uint8
```

2. Change the tensor data type:

```
x=x.type(torch.float)
print(x.dtype)
torch.float32
```

In the next section, we will show you how to convert tensors into NumPy arrays.

Converting tensors into NumPy arrays

We can easily convert PyTorch tensors into NumPy arrays. Let's take a look:

1. Define a tensor:

```
x=torch.rand(2,2)
print(x)
print(x.dtype)

tensor([[0.8074, 0.5728],
        [0.2549, 0.2832]])
torch.float32
```

2. Convert the tensor into a NumPy array:

```
y=x.numpy()
print(y)
print(y.dtype)

[[0.80745    0.5727562 ]
 [0.25486636 0.28319395]]
dtype('float32')
```

In the next section, we will show you how to convert NumPy arrays into tensors.

Converting NumPy arrays into tensors

We can also convert NumPy arrays into PyTorch tensors:

1. Define a NumPy array:

```
import numpy as np
x=np.zeros((2,2),dtype=np.float32)
print(x)
print(x.dtype)

[[0.  0.]
```

```
[0. 0.]
float32
```

2. Convert the NumPy array into a PyTorch tensor:

```
y=torch.from_numpy(x)
print(y)
print(y.dtype)

tensor([[0., 0.],
        [0., 0.]])
torch.float32
```

In the next section, we will show you how to move tensors between devices.

Moving tensors between devices

By default, PyTorch tensors are stored on the CPU. PyTorch tensors can be utilized on a GPU to speed up computing. This is the main advantage of tensors compared to NumPy arrays. To get this advantage, we need to move the tensors to the CUDA device. We can move tensors onto any device using the `.to` method:

1. Define a tensor on CPU:

```
x=torch.tensor([1.5, 2])
print(x)
print(x.device)

x=torch.tensor([1.5, 2.])
cpu
```

2. Define a CUDA device:

```
# define a cuda/gpu device
if torch.cuda.is_available():
    device = torch.device("cuda:0")
```

3. Move the tensor onto the CUDA device:

```
x = x.to(device)
print(x)
print(x.device)

tensor([1.5, 2.], device='cuda:0')
cuda:0
```

4. Similarly, we can move tensors to CPU:

```
# define a cpu device
device = torch.device("cpu")
x = x.to(device)
print(x)
print(x.device)

tensor([1.5, 2.])
cpu
```

5. We can also directly create a tensor on any device:

```
# define a tensor on device
device = torch.device("cuda:0")
x = torch.ones(2,2, device=device)
print(x)

tensor([[1., 1.],
       [1., 1.]], device='cuda:0'
```

In the next section, we will explain each step in detail.

How it works...

First, we defined a tensor, obtained the tensor type, and changed its type. Then, we converted PyTorch tensors into NumPy arrays and vice versa. We also moved tensors between the CPU and CUDA devices. Next, we showed you how to change a tensor data type using the `.type` method. Then, we showed how to convert PyTorch tensors into NumPy arrays using the `.numpy` method.

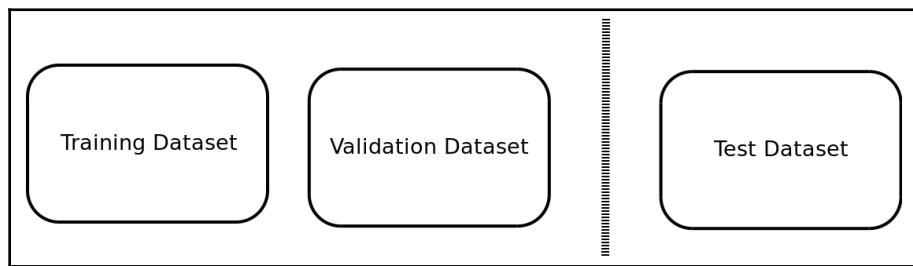
After that, we showed you how to convert a NumPy array into a PyTorch tensor using the `.from_numpy(x)` method. Then, we showed you how to move tensors from a CPU device to a GPU device and vice versa, using the `.to` method. As you have seen, if you do not specify the device, the tensor will be hosted on the CPU device.

See also

Visit the following link for a complete list of Tensor types and operations: <https://pytorch.org/docs/stable/tensors.html>.

Loading and processing data

In most cases, it's assumed that we receive data in three groups: training, validation, and test. We use the training dataset to train the model. The validation dataset is used to track the model's performance during training. We use the test dataset for the final evaluation of the model. The target values of the test dataset are usually hidden from us. We need at least one training dataset and one validation dataset to be able to develop and train a model. Sometimes, we receive only one dataset. In such cases, we can split the dataset into two or three groups, as shown in the following diagram:



Each dataset is comprised of inputs and targets. It is common to represent the inputs with `x` or `X` and the targets with `y` or `Y`. We add the suffixes `train`, `val`, and `test` to distinguish each dataset.

In this recipe, we will learn about PyTorch data tools. We can use these tools to load and process data.

How to do it...

In the following sections, you will learn how to use PyTorch packages to work with datasets.

Loading a dataset

The PyTorch `torchvision` package provides multiple popular datasets.

Let's load the MNIST dataset from `torchvision`:

1. First, we will load the MNIST training dataset:

```
from torchvision import datasets  
  
# path to store data and/or load from  
path2data="../data"  
  
# loading training data  
train_data=datasets.MNIST(path2data, train=True, download=True)
```

2. Then, we will extract the input data and target labels:

```
# extract data and targets  
x_train, y_train=train_data.data, train_data.targets  
print(x_train.shape)  
print(y_train.shape)  
  
torch.Size([60000, 28, 28])  
torch.Size([60000])
```

3. Next, we will load the MNIST test dataset:

```
# loading validation data  
val_data=datasets.MNIST(path2data, train=False, download=True)
```

4. Then, we will extract the input data and target labels:

```
# extract data and targets  
x_val,y_val=val_data.data, val_data.targets  
print(x_val.shape)  
print(y_val.shape)  
  
torch.Size([10000, 28, 28])  
torch.Size([10000])
```

5. After that, we will add a new dimension to the tensors:

```
# add a dimension to tensor to become B*C*H*W
if len(x_train.shape)==3:
    x_train=x_train.unsqueeze(1)
print(x_train.shape)

if len(x_val.shape)==3:
    x_val=x_val.unsqueeze(1)
print(x_val.shape)

torch.Size([60000, 1, 28, 28])
torch.Size([10000, 1, 28, 28])
```

Now, let's display a few sample images.

6. Next, we will import the required packages:

```
from torchvision import utils
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

7. Then, we will define a helper function to display tensors as images:

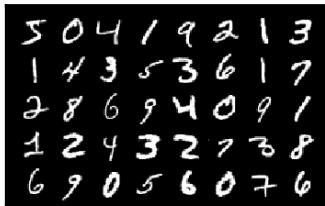
```
def show(img):
    # convert tensor to numpy array
    npimg = img.numpy()
    # Convert to H*W*C shape
    npimg_tr=np.transpose(npimg, (1,2,0))
    plt.imshow(npimg_tr,interpolation='nearest')
```

8. Next, we will create a grid of images and display them:

```
# make a grid of 40 images, 8 images per row
x_grid=utils.make_grid(x_train[:40], nrow=8, padding=2)
print(x_grid.shape)

# call helper function
show(x_grid)
```

The results are shown in the following image:



In the next section, we will show you how to use data transformations together with a dataset.

Data transformation

Image transformation (also called augmentation) is an effective technique that's used to improve a model's performance. The `torchvision` package provides common image transformations through the `transform` class. Let's take a look:

1. Let's define a transform class in order to apply some image transformations on the MNIST dataset:

```
from torchvision import transforms

# loading MNIST training dataset
train_data=datasets.MNIST(path2data, train=True, download=True)

# define transformations
data_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=1),
    transforms.RandomVerticalFlip(p=1),
    transforms.ToTensor(),
])
```

2. Let's apply the transformations on an image from the MNIST dataset:

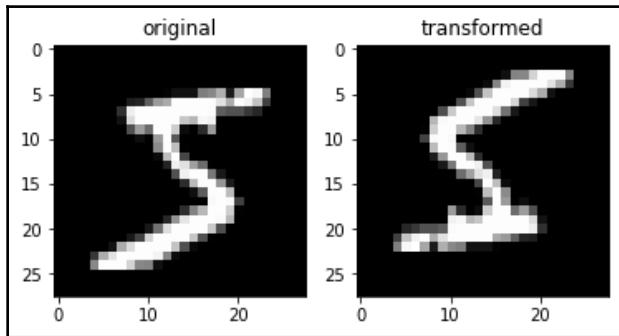
```
# get a sample image from training dataset
img = train_data[0][0]

# transform sample image
img_tr=data_transform(img)

# convert tensor to numpy array
img_tr_np=img_tr.numpy()
```

```
# show original and transformed images
plt.subplot(1,2,1)
plt.imshow(img, cmap="gray")
plt.title("original")
plt.subplot(1,2,2)
plt.imshow(img_tr_np[0], cmap="gray");
plt.title("transformed")
```

The results are shown in the following image:



3. We can also pass the transformer function to the dataset class:

```
# define transformations
data_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(1),
    transforms.RandomVerticalFlip(1),
    transforms.ToTensor(),
])

# Loading MNIST training data with on-the-fly transformations
train_data=datasets.MNIST(path2data, train=True, download=True,
transform=data_transform )
```

In the next section, we will show you how to create a dataset from tensors.

Wrapping tensors into a dataset

If your data is available in tensors, you can wrap them as a PyTorch dataset using the `TensorDataset` class. This will make it easier to iterate over data during training. Let's get started:

1. Let's create a PyTorch dataset by wrapping `x_train` and `y_train`:

```
from torch.utils.data import TensorDataset

# wrap tensors into a dataset
train_ds = TensorDataset(x_train, y_train)
val_ds = TensorDataset(x_val, y_val)

for x,y in train_ds:
    print(x.shape,y.item())
    break

torch.Size([1, 28, 28]) 5
```

In the next section, we will show you how to define a data loader.

Creating data loaders

To easily iterate over the data during training, we can create a data loader using the `DataLoader` class, as follows:

1. Let's create two data loaders for the training and validation datasets:

```
from torch.utils.data import DataLoader

# create a data loader from dataset
train_dl = DataLoader(train_ds, batch_size=8)
val_dl = DataLoader(val_ds, batch_size=8)

# iterate over batches
for xb,yb in train_dl:
    print(xb.shape)
    print(yb.shape)
    break

torch.Size([8, 1, 28, 28])
torch.Size([8])
```

In the next section, we will explain each step in detail.

How it works...

First, we imported the `datasets` package from `torchvision`. This package contains several famous datasets, including MNIST. Then, we downloaded the MNIST training dataset into a local folder. Once downloaded, you can set the `download` flag to `False` in future runs. Next, we extracted the input data and target labels into PyTorch tensors and printed their size. Here, the training dataset contains 60,000 inputs and targets. Then, we repeated the same step for the MNIST test dataset. To download the MNIST test dataset, we set the `train` flag to `False`. Here, the test dataset contains 10,000 inputs and targets.

Next, we added a new dimension to the input tensors since we want the tensor shape to be $B*C*H*W$, where B , C , H , and W are batch size, channels, height, and width, respectively. This is the common shape for the inputs tensors in PyTorch. Then, we defined a helper function to display sample images. We used `utils` from `torchvision` to create a grid of 40 images in five rows and eight columns.

In the *Data transformation* subsection, we introduced the `torchvision.transforms` package. This package provides multiple transformation functions. We composed the `RandomHorizontalFlip` and `RandomVerticalFlip` methods to augment the dataset and the `ToTensor` method to convert images into PyTorch tensors. The probability of horizontal and vertical flips was set to $p=1$ to enforce flipping in the next step. We employed the data transformer on a sample image. Check out the original and the transformed image. The transformed image has been flipped both vertically and horizontally.

Then, we passed the transformer function to the dataset class. This way, data transformation will happen on-the-fly. This is a useful technique for large datasets that cannot be loaded into memory all at once.

In the *Wrapping tensors into a dataset* subsection, we created a dataset from tensors. For example, we can create a PyTorch dataset by wrapping `x_train` and `y_train`. This technique will be useful for cases where the input and output data is available as tensors.

In the *Creating data loaders* subsection, we used the `DataLoader` class to define data loaders. This is a good technique to easily iterate over datasets during training or evaluation. When creating a data loader, we need to specify the batch size. We created two data loaders from `train_ds` and `val_ds`. Then, we extracted a mini-batch from `train_dl`. Check out the shape of the mini-batch.

Building models

A model is a collection of connected layers that process the inputs to generate the outputs. You can use the `nn` package to define models. The `nn` package is a collection of modules that provide common deep learning layers. A module or layer of `nn` receives input tensors, computes output tensors, and holds the weights, if any. There are two methods we can use to define models in PyTorch: `nn.Sequential` and `nn.Module`.

How to do it...

We will define a linear layer, a two-layer network, and a multilayer convolutional network.

Defining a linear layer

Let's create a linear layer and print out its output size:

```
from torch import nn

# input tensor dimension 64*1000
input_tensor = torch.randn(64, 1000)

# linear layer with 1000 inputs and 100 outputs
linear_layer = nn.Linear(1000, 100)

# output of the linear layer
output = linear_layer(input_tensor)
print(output.size())
```

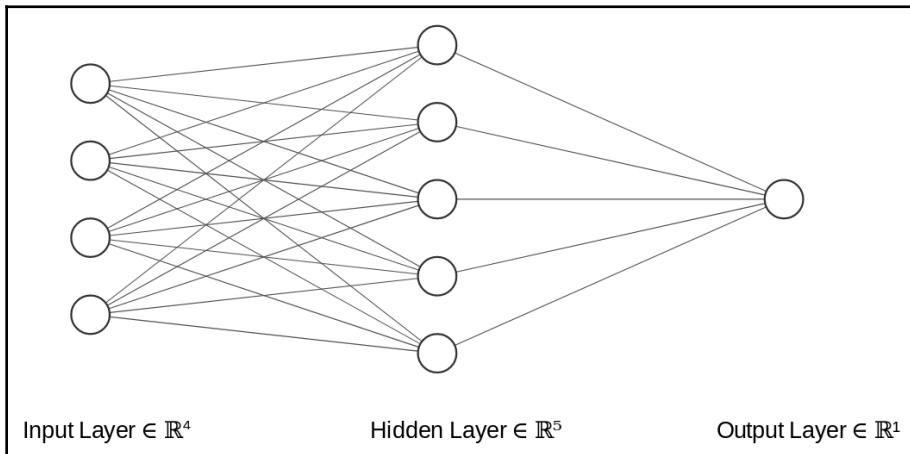
The following code will print out its output size:

```
torch.Size([64, 100])
```

In the next section, we will show you how to define a model using the `nn.Sequential` package.

Defining models using nn.Sequential

We can use the `nn.Sequential` package to create a deep learning model by passing layers in order. Consider the two-layer neural network depicted in the following image:



As we can see, the network has four nodes as input, five nodes in the hidden layer, and one node as the output. Next, we will show you how to implement the network:

1. Let's implement and print the model using `nn.Sequential`:

```
from torch import nn

# define a two-layer model
model = nn.Sequential(
    nn.Linear(4, 5),
    nn.ReLU(),
    nn.Linear(5, 1),
)
print(model)
```

The output of the preceding code is as follows:

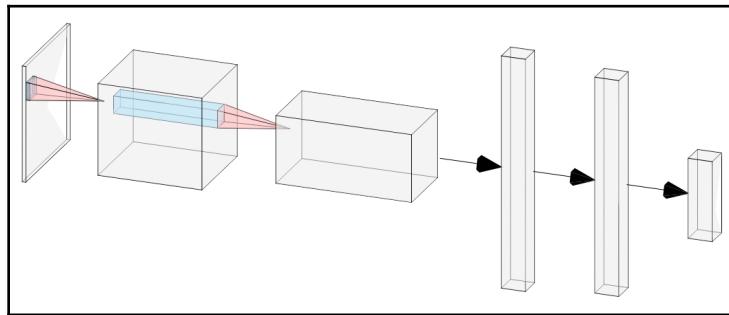
```
Sequential(
  (0): Linear(in_features=4, out_features=5, bias=True)
  (1): ReLU()
  (2): Linear(in_features=5, out_features=1, bias=True)
)
```

In the next section, we will introduce another way of defining a model.

Defining models using nn.Module

Another way of defining models in PyTorch is by subclassing the `nn.Module` class. In this method, we specify the layers in the `__init__` method of the class. Then, in the `forward` method, we apply the layers to inputs. This method provides better flexibility for building customized models.

Consider a multilayer model, as shown in the following image:



As seen in the preceding image, the model has two convolutional layers and two fully connected layers. Next, we will show you how to implement the model.

Let's implement the multilayer model using `nn.Module`:

1. First, we will implement the bulk of the class:

```
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

    def forward(self, x):
        pass
```

2. Then, we will define the `__init__` function:

```
def __init__(self):
    super(Net, self).__init__()
    self.conv1 = nn.Conv2d(1, 20, 5, 1)
    self.conv2 = nn.Conv2d(20, 50, 5, 1)
    self.fc1 = nn.Linear(4*4*50, 500)
    self.fc2 = nn.Linear(500, 10)
```

3. Next, we will define the `forward` function:

```
def forward(self, x):
    x = F.relu(self.conv1(x))
    x = F.max_pool2d(x, 2, 2)
    x = F.relu(self.conv2(x))
    x = F.max_pool2d(x, 2, 2)
    x = x.view(-1, 4*4*50)
    x = F.relu(self.fc1(x))
    x = self.fc2(x)
    return F.log_softmax(x, dim=1)
```

4. Then, we will override both class functions, `__init__` and `forward`:

```
Net.__init__ = __init__
Net.forward = forward
```

5. Next, we will create an object of the `Net` class and print the model:

```
model = Net()
print(model)
```

The output of the preceding code is as follows:

```
Net(
  (conv1): Conv2d(1, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(20, 50, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=800, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=10, bias=True)
)
```

In the next section, we will show you how to move the model to a CUDA device.

Moving the model to a CUDA device

A model is a collection of parameters. By default, the model will be hosted on the CPU:

1. Let's get the model's device:

```
print(next(model.parameters()).device)
```

The preceding snippet will print the following output:

```
cpu
```

2. Then, we will move the model to the CUDA device:

```
device = torch.device("cuda:0")
model.to(device)
print(next(model.parameters()).device)
cuda:0
```

In the next section, we will show you how to print the model summary.

Printing the model summary

It is usually helpful to get a summary of the model to see the output shape and the number of parameters in each layer. Printing a model does not provide this kind of information. We can use the `torchsummary` package from the following GitHub repository for this purpose: <https://github.com/sksq96/pytorch-summary>. Let's get started:

1. Install the `torchsummary` package:

```
pip install torchsummary
```

2. Let's get the model summary using `torchsummary`:

```
from torchsummary import summary
summary(model, input_size=(1, 28, 28))
```

The preceding code will display the model summary when it's executed:

```
-----
      Layer (type)          Output Shape       Param #
=====
Conv2d-1           [-1, 20, 24, 24]           520
Conv2d-2           [-1, 50, 8, 8]            25,050
Linear-3           [-1, 500]                400,500
Linear-4           [-1, 10]                 5,010
=====
Total params: 431,080
Trainable params: 431,080
Non-trainable params: 0
-----
Input size (MB): 0.00
Forward/backward pass size (MB): 0.12
Params size (MB): 1.64
Estimated Total Size (MB): 1.76
-----
```

In the next section, we will explain each step in detail.

How it works...

First, we showed you how to create a linear layer using the `nn` package. The linear layer receives the input of the 64×1000 dimension, holds the weights of the 1000×100 dimension, and computes the output of the 64×100 dimension.

Next, we defined a two-layer neural network using `nn.Sequential`. There were four neurons in the input layer, five neurons in the hidden layer, and one neuron in the output layer. Using the `print` command, you can visualize the model's layers.

Next, we defined a multilayer model using `nn.Module`. The model has two `Conv2d` layers and two fully connected linear layers. For better code readability, we presented the `Net` class in a few snippets. First, we defined the bulk of the class. Then, we defined the `__init__` function. As you saw, two `Conv2d` layers and two linear layers were defined in this function. Next, we defined the `forward` function. In this function, we defined the outline of the model and the way layers are connected to each other.

We used `relu` and `max_pool2d` from `torch.nn.functional` to define the activation function and pooling layers, respectively. Check out the way we used the `.view` method to flatten the extracted features from the `Conv2d` layers. The feature size was 4×4 and there were 50 channels in the `self.conv2` layer. Due to this, the flatten size is $50 \times 4 \times 4$. Also, check out the returned values from the `forward` function. As we saw, the `log_softmax` function was applied to the outputs. Next, we overrode the `Net` class functions. Finally, we created an object of the `Net` class and called it `model`. Then, we printed the model. Note that the `print` command does not show functional layers such as `relu` and `max_pool2d`.

In the *Moving the model to a CUDA device* subsection, we verified that the model was hosted on the CPU device. Then, we moved the model to the CUDA device using the `.to` method. Here, we moved the first GPU or "`cuda:0`". If your system is equipped with multiple GPU devices, you can select a different number, for instance, "`cuda:2`".

Next, we installed the `torchsummary` package in the conda environment using the provided command.

If you do not want to install this package, the other option is to copy `torchsummary.py` into the folder of your code.

To get a model summary using `torchsummary`, we need to pass the input dimension to the `summary` function. For our MNIST example, we passed `(1, 28, 28)` as the input dimension and displayed the model summary. As seen, the output shape and the number of parameters of each layer, except functional layers, is shown in the summary.

Finally, we got the model summary using the `torchsummary` package.

Defining the loss function and optimizer

The loss function computes the distance between the model outputs and targets. It is also called the objective function, cost function, or criterion. Depending on the problem, we will define the appropriate loss function. For instance, for classification problems, we usually define the cross-entropy loss.

We use the optimizer to update the model parameters (also called weights) during training. The `optim` package in PyTorch provides implementations of various optimization algorithms. These include **stochastic gradient descent (SGD)** and its variants, that is, Adam, RMSprop, and so on.

How to do it...

In this section, we will look at defining the loss function and optimizer in PyTorch.

Defining the loss function

We will define a loss function and test it on a mini-batch. Let's get started:

1. First, we will define the negative log-likelihood loss:

```
from torch import nn
loss_func = nn.NLLLoss(reduction="sum")
```

2. Let's test the loss function on a mini-batch:

```
for xb, yb in train_dl:
    # move batch to cuda device
    xb=xb.type(torch.float).to(device)
    yb=yb.to(device)
    # get model output
    out=model(xb)
    # calculate loss value
    loss = loss_func(out, yb)
    print (loss.item())
    break
```

The preceding snippet will print the following output:

```
69.37257385253906
```

3. Let's compute the gradients with respect to the model parameters:

```
# compute gradients
loss.backward()
```

In the next step, we will show you how to define an optimizer.

Defining the optimizer

We will define the optimizer and present the steps backward. Let's get started:

1. Let's define the Adam optimizer:

```
from torch import optim
opt = optim.Adam(model.parameters(), lr=1e-4)
```

2. Use the following code to update the model parameters:

```
# update model parameters
opt.step()
```

3. Next, we set the gradients to zero:

```
# set gradients to zero
opt.zero_grad()
```

In the next section, we will explain each step in detail.

How it works...

First, we defined the loss function. We used the `torch.nn` package to define the negative log-likelihood loss. This loss is useful for training a classification problem with multiple classes. The input to this loss function should be log-probabilities. If you recall from the *Building models* section, we applied `log_softmax` at the output layer to get log-probabilities from the model. Next, we presented the forward path. We extracted a mini-batch, fed it to the model, and calculated the loss value. Next, we used the `.backward` method to compute the gradients of the loss with respect to the model parameters. This step will be used during the backpropagation algorithm.

Next, we define the `Adam` optimizer. The inputs to the optimizer are the model parameters and the learning rate. Then, we presented the `.step()` model to automatically update the model parameters. Don't forget to set the gradients to zero before computing the gradients of the next batch.

See also

The `torch.nn` package provides several common loss functions. For a list of supported loss functions, please visit the following link: <https://pytorch.org/docs/stable/nn.html>.

For more information on the `torch.optim` package, please visit the following link: <https://pytorch.org/docs/stable/optim.html>.

Training and evaluation

Once all the ingredients are ready, we can start training the model. In this recipe, you will learn how to properly train and evaluate a deep learning model.

How to do it...

We will develop helper functions for batch and epoch processing and training the model. Let's get started:

1. Let's develop a helper function to compute the loss value per mini-batch:

```
def loss_batch(loss_func, xb, yb, yb_h, opt=None):
    # obtain loss
    loss = loss_func(yb_h, yb)
    # obtain performance metric
    metric_b = metrics_batch(yb, yb_h)
    if opt is not None:
        loss.backward()
        opt.step()
        opt.zero_grad()

    return loss.item(), metric_b
```

2. Next, we will define a helper function to compute the accuracy per mini-batch:

```
def metrics_batch(target, output):
    # obtain output class
    pred = output.argmax(dim=1, keepdim=True)
    # compare output class with target class
    corrects=pred.eq(target.view_as(pred)).sum().item()
    return corrects
```

3. Next, we will define a helper function to compute the loss and metric values for a dataset:

```
def loss_epoch(model, loss_func, dataset_dl, opt=None):
    loss=0.0
    metric=0.0
    len_data=len(dataset_dl.dataset)
    for xb, yb in dataset_dl:
        xb=xb.type(torch.float).to(device)
        yb=yb.to(device)
        # obtain model output
        yb_h=model(xb)

        loss_b,metric_b=loss_batch(loss_func, xb, yb,yb_h, opt)
        loss+=loss_b
        if metric_b is not None:
            metric+=metric_b
    loss/=len_data
    metric/=len_data
    return loss, metric
```

4. Finally, we will define the `train_val` function:

```
def train_val(epochs, model, loss_func, opt, train_dl, val_dl):
    for epoch in range(epochs):
        model.train()
        train_loss,
        train_metric=loss_epoch(model,loss_func,train_dl,opt)
        model.eval()
        with torch.no_grad():
            val_loss, val_metric=loss_epoch(model,loss_func,val_dl)
            accuracy=100*val_metric

            print("epoch: %d, train loss: %.6f, val loss: %.6f,
accuracy: %.2f" %(epoch, train_loss,val_loss,accuracy))
```

5. Let's train the model for a few epochs:

```
# call train_val function
num_epochs=5
train_val(num_epochs, model, loss_func, opt, train_dl, val_dl)
```

Training will start and you should see its progress, as shown in the following code:

```
epoch: 0, train loss: 0.294502, val loss: 0.093089, accuracy: 96.94
epoch: 1, train loss: 0.080617, val loss: 0.061121, accuracy: 98.06
epoch: 2, train loss: 0.050562, val loss: 0.049555, accuracy: 98.49
epoch: 3, train loss: 0.035071, val loss: 0.049693, accuracy: 98.45
epoch: 4, train loss: 0.025703, val loss: 0.050179, accuracy: 98.49
```

In the next section, we will show you how to store and load a model.

Storing and loading models

Once training is complete, we'll want to store the trained parameters in a file for deployment and future use. There are two ways of doing so.

Let's look at the first method:

1. First, we will store the model parameters or `state_dict` in a file:

```
# define path2weights
path2weights="../models/weights.pt"

# store state_dict to file
torch.save(model.state_dict(), path2weights)
```

2. To load the model parameters from the file, we will define an object of the `Net` class:

```
# define model: weights are randomly initiated
model = Net()
```

3. Then, we will load `state_dict` from the file:

```
weights=torch.load(path2weights)
```

4. Next, we will set `state_dict` to the model:

```
model.load_state_dict(weights)
```

Now, let's look at the second method:

1. First, we will store the model in a file:

```
# define a path2model
path2model=("./models/model.pt"

# store model and weights into a file
torch.save(model,path2model)
```

2. To load the model parameters from the file, we will define an object of the Net class:

```
# define model: weights are randomly initiated
_model = Net()
```

3. Then, we will load the model from the local file:

```
_model=torch.load(path2model)
```

In the next section, we will show you how to deploy the model.

Deploying the model

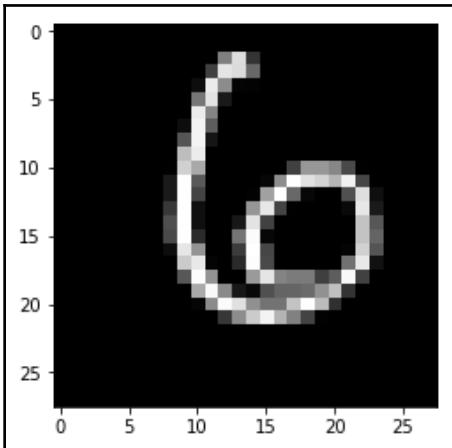
To deploy a model, we need to load the model using the methods described in the previous section. Once the model has been loaded into memory, we can pass new data to the model. Let's get started:

1. To deploy the model on a sample image from the validation dataset, we will get a sample tensor:

```
n=100
x= x_val[n]
y=y_val[n]
print(x.shape)
plt.imshow(x.numpy() [0],cmap="gray")

torch.Size([1, 28, 28])
```

The sample image is shown in the following screenshot:



2. Then, we will preprocess the tensor:

```
# we use unsqueeze to expand dimensions to 1*C*H*W
x= x.unsqueeze(0)

# convert to torch.float32
x=x.type(torch.float)

# move to cuda device
x=x.to(device)
```

3. Next, we will get the model prediction:

```
# get model output
output=_model(x)

# get predicted class
pred = output.argmax(dim=1, keepdim=True)
print (pred.item(),y.item())
```

The model prediction and the ground truth label are printed out after you execute the preceding code:

6 6

In the next section, we will explain each step in detail.

How it works...

First, we developed a helper function to compute the loss and metric value per mini-batch. The `opt` argument of the function refers to the optimizer. If given, the gradients are computed and the model parameters are updated per mini-batch.

Next, we developed a helper function to compute a performance metric. The performance metric can be defined depending on the task. Here, we chose the accuracy metric for our classification task. We used `output.argmax` to get the predicted class with the highest probability.

Next, we defined a helper function to compute the loss and metric values for an entire dataset. We used the data loader object to get mini-batches, feed them to the model, and compute the loss and metrics per mini-batch. We used two running variables to add loss and metric values.

Next, we defined a helper function to train the model for multiple epochs. In each epoch, we also evaluated the model's performance using the validation dataset. Note that we set the model in training and evaluation modes using `model.train()` and `model.eval()`, respectively. Moreover, we used `torch.no_grad()` to stop autograd from calculating the gradients during evaluation.

Next, we explored two methods of storing the trained model. In the first method, we stored `state_dict` or model parameters only. Whenever we need the trained model for deployment, we have to create an object of the model, then load the parameters from the file, and then set the parameters to the model. This is the recommended method by PyTorch creators.

In the second method, we stored the model into a file. In other words, we stored both the model and `state_dict` into one file. Whenever we need the trained model for deployment, we need to create an object of the `Net` class. Then, we loaded the model from the file. So, there is no actual benefit of doing this compared to the previous method.

Next, we deployed the model on a sample image of the validation dataset. The sample image shape is `C*H*W`. Thus, we added a new dimension to become `1*C*H*W`. Then, we converted the tensor type into `torch.float32` and moved it to a CUDA device.



Make sure that the model and data are hosted on the same device at deployment, otherwise, you will encounter an error.

There's more...

Training deep learning models requires developing intuitions. We will introduce other techniques such as early stopping and learning rate schedules to avoid overfitting and improve performance in the next chapter.

2

Binary Image Classification

Image classification (also called **image recognition**) is an important task in computer vision. In this task, we assume that images contain one main object. Here, our goal is to classify the main object. There are two types of image classification: binary classification and multi-class classification. In this chapter, we will develop a deep learning model using PyTorch to perform binary classification on images.

The goal of binary image classification is to classify images into two categories. For instance, we may want to know if a medical image is normal or malignant. The images could be grayscale with one channel or color image with three channels.

In this chapter, we'll also learn how to create an algorithm to identify metastatic cancer in small image patches taken from larger digital pathology scans.

In particular, we will cover the following recipes:

- Exploring the dataset
- Creating a custom dataset
- Splitting the dataset
- Transforming the data
- Creating dataloaders
- Building the classification model
- Defining the loss function
- Defining the optimizer
- Training and evaluation of the model
- Deploying the model
- Model inference on test data

Exploring the dataset

We'll use the dataset provided in the Histopathologic Cancer Detection competition on Kaggle. The goal of this competition is to classify image patches as normal or malignant. In this section, we will investigate the dataset.

Getting ready

You need to create an account on the Kaggle website, join the competition, and agree with the Kaggle terms to be able to download the dataset. After signing up, visit the following link and download the competition dataset: <https://www.kaggle.com/c/histopathologic-cancer-detection/data>.

After downloading it, extract the ZIP files into a folder named `data`. We recommend putting the `data` folder in the same location as your code for ease of reference.

Inside the `data` folder, there are two folders: `train` and `test`. The `train` folder contains 220,025 `.tif` images that are 96x96 in size. The `.tif` images are named with an image ID. The `train_labels.csv` file provides the ground truth for the images in the `train` folder.

There are no labels for the test images. If we were to compete in the competition, we would need to provide predictions on the test dataset and submit them to the competition for evaluation.

How to do it...

To learn about the dataset, we'll start by loading the labels and displaying a few sample images from the dataset:

1. Let's begin by reading `train_labels.csv` and printing out its head:

```
import pandas as pd  
  
path2csv=".\\data\\train_labels.csv"  
labels_df=pd.read_csv(path2csv)  
labels_df.head()
```

The following table shows the top five rows of the CSV file:

	id	label
0	f38a6374c348f90b587e046aac6079959adf3835	0
1	c18f2d887b7ae4f6742ee445113fa1aef383ed77	1
2	755db6279dae599eb84d39a9123cce439965282d	0
3	bc3f0c64fb968ff4a8bd33af6971ecae77c75e08	0
4	068aba587a4950175d04c680d38943fd488d6a9d	0

2. Let's count the number of normal and malignant cases:

```
print(labels_df['label'].value_counts())
```

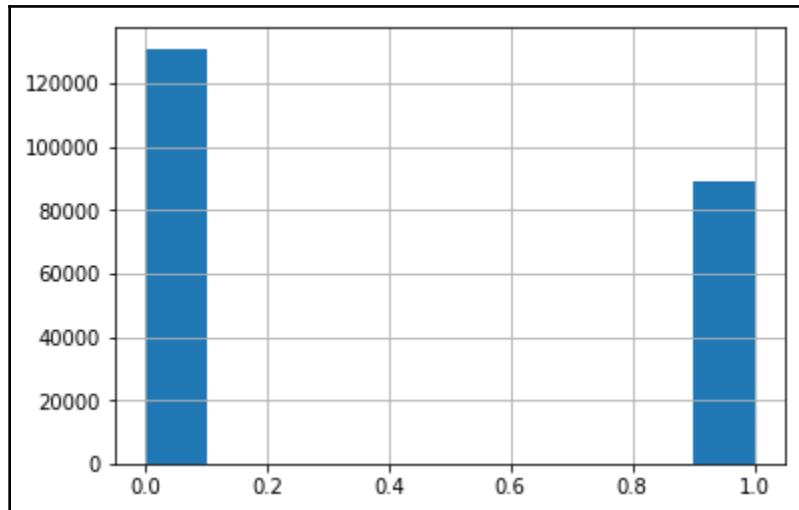
The preceding code snippet will print the following output:

```
0    130908  
1    89117  
Name: label, dtype: int64
```

3. Let's look at a histogram of the labels:

```
%matplotlib inline  
labels_df['label'].hist();
```

The following screenshot shows a histogram of the labels:



4. Let's visualize a few images that have a positive label. A positive label shows that the center 32×32 region of an image contains at least one pixel of tumor tissue. First, we import the required packages:

```
import matplotlib.pyplot as plt
from PIL import Image, ImageDraw
import numpy as np
import os
%matplotlib inline
```

Then, we get the IDs of the malignant images:

```
# get ids for malignant images
malignantIds = labels_df.loc[labels_df['label']==1]['id'].values
```

Define the path to data:

```
# data is stored here
path2train="../data/train/"
```

Next, we define a flag to show images in grayscale or color mode:

```
# show images in grayscale, if you want color change it to True
color=False
```

Next, we set the figure sizes:

```
plt.rcParams['figure.figsize'] = (10.0, 10.0)
plt.subplots_adjust(wspace=0, hspace=0)
nrows, ncols=3,3
```

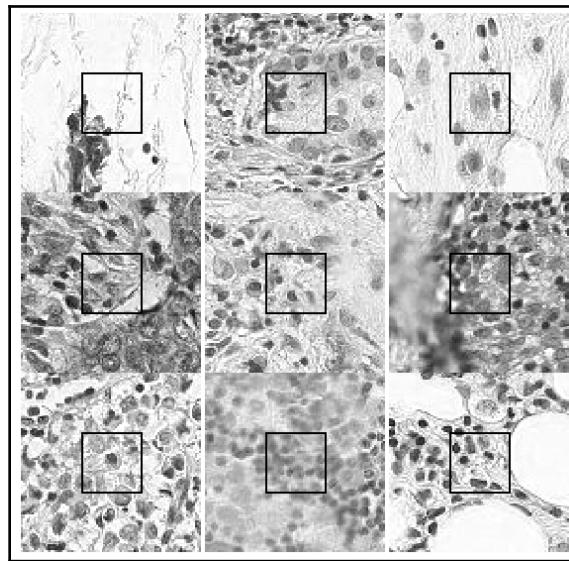
Next, we display the images:

```
for i,id_ in enumerate(malignantIds[:nrows*ncols]):
    full_filenames = os.path.join(path2train , id_ +'.tif')

    # load image
    img = Image.open(full_filenames)

    # draw a 32*32 rectangle
    draw = ImageDraw.Draw(img)
    draw.rectangle(((32, 32), (64, 64)), outline="green")
    plt.subplot(nrows, ncols, i+1)
    if color is True:
        plt.imshow(np.array(img))
    else:
        plt.imshow(np.array(img) [:,:,0], cmap="gray")
    plt.axis('off')
```

Some sample positive images are shown in the following screenshot:



5. Let's also get the image's shape and minimum and maximum pixel values:

```
print("image shape:", np.array(img).shape)
print("pixel values range from %s to %s" %(np.min(img),
np.max(img)))

image shape: (96, 96, 3)
pixel values range from 0 to 255
```

In the next section, we will go through each step in detail.

How it works...

First, we loaded the labels as a DataFrame using pandas. The DataFrame has two columns: **id** and **label**. The **id** column refers to the image filename, while the **label** column shows whether the image is normal (**label=0**) or malignant (**label=1**).

In *step 2*, we counted the number of normal and malignant images. As we saw, around 59% of the images were normal and 41% were malignant. The histogram in *step 3* shows somewhat imbalanced data.

Then, we displayed a few sample images. Please note that we show the images in grayscale mode as they might be unpleasant in color.

As we saw, the images are colored images with three channels (96, 96, 3). The pixel values are in the range [0, 255].

It is really difficult to tell whether an image is malignant just by looking at it. So, you can imagine why it's a difficult task for clinicians to inspect so many images every day. Hopefully, we can develop an automated tool to help them in their routine tasks.

Creating a custom dataset

A traditional method for working with a dataset would be to load all images into NumPy arrays. Since we are dealing with a relatively large dataset, this would be a waste of our computer resources. If you are short on RAM, this would be impossible. Luckily, PyTorch has a powerful tool to handle large datasets.

We can create a custom `Dataset` class by subclassing the PyTorch `Dataset` class. When creating a custom `Dataset` class, make sure to define two essential functions: `__len__` and `__getitem__`. The `__len__` function returns the dataset's length. This function is callable with the Python `len` function. The `__getitem__` function returns an image at the specified index.

How to do it...

We will define a class for the custom dataset, define the transformation function, and then load an image from the dataset using the `Dataset` class. Let's get started:

1. Let's create a PyTorch custom dataset for our data:

First, we will load the required packages.

```
from PIL import Image
import torch
from torch.utils.data import Dataset
import pandas as pd
import torchvision.transforms as transforms
import os
```

Don't forget to fix the random seed for reproducibility:

```
# fix torch random seed
torch.manual_seed(0)
```

Define the `histoCancerDataset` class:

```
class histoCancerDataset(Dataset):
    def __init__(self, data_dir, transform, data_type="train"):
        # path to images
        path2data=os.path.join(data_dir,data_type)

        # get a list of images
        filenames = os.listdir(path2data)

        # get the full path to images
        self.full_filenames = [os.path.join(path2data, f) for f in
        filenames]
```

The class continues with the following code:

```
# labels are in a csv file named train_labels.csv
csv_filename=data_type+"_labels.csv"
path2csvLabels=os.path.join(data_dir,csv_filename)
labels_df=pd.read_csv(path2csvLabels)

# set data frame index to id
labels_df.set_index("id", inplace=True)

# obtain labels from data frame
self.labels = [labels_df.loc[filename[:-4]].values[0] for
filename in filenames]

self.transform = transform
```

The class continues with the following code:

```
def __len__(self):
    # return size of dataset
    return len(self.full_filenames)
def __getitem__(self, idx):
    # open image, apply transforms and return with label
    image = Image.open(self.full_filenames[idx]) # PIL image
    image = self.transform(image)
    return image, self.labels[idx]
```

2. Next, we will define a simple transformation that only converts a PIL image into PyTorch tensors. Later, we will expand this:

```
import torchvision.transforms as transforms
data_transformer = transforms.Compose([transforms.ToTensor()])
```

3. Then, we will define an object of the custom dataset for the `train` folder:

```
data_dir = "./data/"
histo_dataset = histoCancerDataset(data_dir, data_transformer,
"train")
print(len(histo_dataset))
220025
```

4. Next, we will load an image using the custom dataset:

```
# load an image
img,label=histo_dataset[9]
print(img.shape,torch.min(img),torch.max(img))

torch.Size([3, 96, 96]) tensor(0.) tensor(1.)
```

In the next section, we will go through each step in detail.

How it works...

In *step 1*, we imported the required packages. The PIL package is imported to load images. From `torch.utils.data`, we import `Dataset` as the base class of our custom dataset. The `pandas` package is imported to load the CSV files. We also used `torchvision` for data transformation.

Then, we defined the `Dataset` class. In the `__init__` function, we received the path to the data and the `data_type`. The `data_type` can be either `train` or `test`. The class, like any PyTorch dataset, has the `__len__` and `__getitem__` functions. The `__len__` function returns the length of the dataset. The `__getitem__` function also returns the transformed image at the given index and its corresponding label.

In *step 2*, we defined the transformation function. For now, we only convert PIL images into Pytorch tensors in the transformation function.

In *step 3*, as expected, the length of `histo_dataset` is 220025.

In *step 4*, we can see that the dataset returns images in the (Channels, Height, Width) format and pixel values are normalized to the range [0.0, 1.0]. This is the result of transforms. `ToTensor()` converts a PIL image into the range [0, 255] to `torch.FloatTensor` of shape (C x H x W) in the range [0.0, 1.0]. It is common to use this formatting when working with images in PyTorch.

Splitting the dataset

We need to provide a validation dataset to track the model's performance during training. We use 20% of `histo_dataset` as the validation dataset and use the rest as the training dataset.

How to do it...

We will split the dataset into training and validation and then display a few sample images from each dataset. Let's get started:

1. Let's split `histo_dataset`:

```
from torch.utils.data import random_split

len_histo=len(histo_dataset)
len_train=int(0.8*len_histo)
len_val=len_histo-len_train

train_ds,val_ds=random_split(histo_dataset,[len_train,len_val])

print("train dataset length:", len(train_ds))
print("validation dataset length:", len(val_ds))

train dataset length: 176020
validation dataset length: 44005
```

2. Then, we can get an image from the training dataset:

```
for x,y in train_ds:
    print(x.shape,y)
    break

torch.Size([3, 96, 96]) 1
```

3. We will also get an image from the validation dataset:

```
for x,y in val_ds:  
    print(x.shape,y)  
    break  
  
torch.Size([3, 96, 96]) 1
```

4. Let's display a few samples from train_ds.

Import the required packages:

```
from torchvision import utils  
import numpy as np  
import matplotlib.pyplot as plt  
%matplotlib inline  
np.random.seed(0)
```

Define a helper function to show an image:

```
def show(img,y,color=False):  
    # convert tensor to numpy array  
    npimg = img.numpy()  
    # Convert to H*W*C shape  
    npimg_tr=np.transpose(npimg, (1,2,0))  
    if color==False:  
        npimg_tr=npimg_tr[:, :, 0]  
        plt.imshow(npimg_tr,interpolation='nearest',cmap="gray")  
    else:  
        # display images  
        plt.imshow(npimg_tr,interpolation='nearest')  
    plt.title("label: "+str(y))
```

Create a grid of sample images:

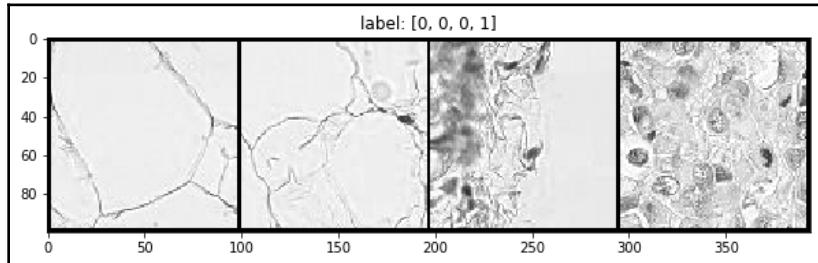
```
grid_size=4  
rnd_inds=np.random.randint(0,len(train_ds),grid_size)  
print("image indices:",rnd_inds)  
  
x_grid_train=[train_ds[i][0] for i in rnd_inds]  
y_grid_train=[train_ds[i][1] for i in rnd_inds]  
  
x_grid_train=utils.make_grid(x_grid_train, nrow=4, padding=2)  
print(x_grid_train.shape)
```

Call the helper function to display the grid:

```
plt.rcParams['figure.figsize'] = (10.0, 5)
show(x_grid_train,y_grid_train)

image indices: [ 43567 173685 117952 152315]
torch.Size([3, 100, 394])
```

Some sample images from `train_ds` are shown in the following image:



5. We can also show a few samples from `val_ds`:

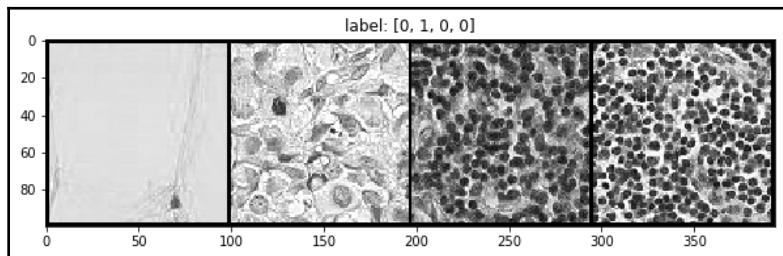
```
grid_size=4
rnd_inds=np.random.randint(0,len(val_ds),grid_size)
print("image indices:",rnd_inds)
x_grid_val=[val_ds[i][0] for i in range(grid_size)]
y_grid_val=[val_ds[i][1] for i in range(grid_size)]

x_grid_val=utils.make_grid(x_grid_val, nrow=4, padding=2)
print(x_grid_val.shape)

show(x_grid_val,y_grid_val)

image indices: [30403 32103 41993 20757]
torch.Size([3, 100, 394])
```

Some sample images from `val_ds` are shown in the following image:



In the next section, we will explain each step in detail.

How it works...

In *step 1*, we used the `random_split` function from `torch.utils.data` to split the dataset. This will return two datasets: `train_ds` and `val_ds` with lengths of 176020 and 44005, respectively.

In *steps 2 and 3*, we get an image and its label from each dataset. The image shape was in the format ($C \times H \times W$), as expected.

In *steps 4 and 5*, we displayed sample images from `train_ds` and `val_ds`.

Transforming the data

Image transformation and image augmentation are necessary for training deep learning models. By using image transformations, we can expand our dataset or resize and normalize it to achieve better model performance. Typical transformations include horizontal and vertical flipping, rotation, and resizing. The good news is that we can use various image transformations for our binary classification model without making label changes. For instance, if we rotate or flip a malignant image, it will remain malignant. In this recipe, you will learn how to use the `torchvision` package to perform on-the-fly image transformation during training.

How to do it...

In this recipe, we will define a few image transformations and then update the dataset transformation function. Let's get started:

1. First, let's define the following transformations for the training dataset:

```
train_transformer = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.5),
    transforms.RandomRotation(45),
    transforms.RandomResizedCrop(96, scale=(0.8, 1.0), ratio=(1.0, 1.0)),
    transforms.ToTensor()])
```

2. For the validation dataset, we don't need any augmentation. So, we only convert the images into tensors in the `transforms` function:

```
val_transformer = transforms.Compose([transforms.ToTensor()])
```

3. After defining the transformations, we overwrite the transform functions of `train_ds` and `val_ds`:

```
# overwrite the transform functions
train_ds.transform=train_transformer
val_ds.transform=val_transformer
```

In the next section, we will explain each step in detail.

How it works...

In *step 1*, `RandomHorizontalFlip` and `RandomVerticalFlip` will flip the image horizontally and vertically with a probability of 0.5, respectively.

The `RandomRotation` function rotates images in the range of [-45,45] degrees.

Also, `RandomSizedCrop` crops a square image randomly in the range of [72, 96] and then resizes it to the original size of 96*96.

In *step 2*, we used `transforms.ToTensor` to normalize the images in the range [0, 1] and convert them into tensors. It is common to normalize images to zero-mean and unit-variance by subtracting the mean and diving into the standard deviation of the pixel values. However, here, we opt not to do any further normalization.

The validation dataset was only normalized to the range [0, 1] without any data augmentation.

Finally, in *step 3*, we defined the transformations and overwrote the transform functions of `train_ds` and `val_ds`.

Creating dataloaders

We are ready to create a PyTorch dataloader. If we do not use dataloaders, we have to write code to loop over datasets and extract a data batch. This process can be made automatically using a PyTorch Dataloader.

How to do it...

In this recipe, we will define two dataloaders and then show you how to get a batch of data using the dataloaders. Let's get started:

1. First, let's define two dataloaders for the datasets:

```
from torch.utils.data import DataLoader
train_dl = DataLoader(train_ds, batch_size=32, shuffle=True)
val_dl = DataLoader(val_ds, batch_size=64, shuffle=False)
```

2. Then, we get a data batch from the training dataloader:

```
# extract a batch from training data
for x, y in train_dl:
    print(x.shape)
    print(y.shape)
    break
torch.Size([32, 3, 96, 96])
torch.Size([32])
```

3. Next, we get a data batch from the validation dataloader:

```
for x, y in val_dl:
    print(x.shape)
    print(y.shape)
    break
```

The preceding snippet will print the following output:

```
torch.Size([64, 3, 96, 96])
torch.Size([64])
```

In the next section, we will explain each step in detail.

How it works...

In *step 1*, when defining a PyTorch Dataloader, we need to define the batch size. The batch size determines the number of images to be extracted from the dataset in each iteration. The typical values of the batch size for classification tasks are in the range of [8-128]. Also, note that we do not need to shuffle the validation data during evaluation.

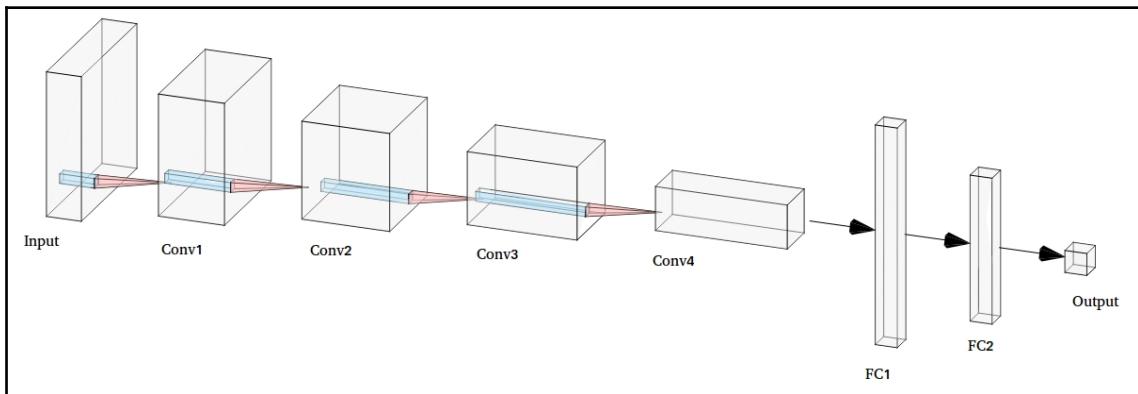


If your computer or the GPU device on your computer does not have enough memory, you can reduce the batch size to avoid memory errors.

When iterating over a dataloader, it will automatically extract batches of data from the dataset. As seen in *step 2*, for the train dataloader, we extracted batches of images that were 32 in size. This was 64 for the validation dataloader since we set the batch size to 64.

Building the classification model

In this section, we will build a model for our binary classification task. Our model is comprised of four **convolutional neural networks** (CNNs) and two fully connected layers, as shown in the following diagram:



As we can see, there are four convolutional layers and two fully connected layers in the model. After each convolutional layer, there is a pooling layer. The pooling layers are not shown in the preceding diagram. The convolutional layers process the input image and extract a feature vector, which is fed to the fully connected layers. There is an output layer for the binary classification.

In this recipe, we will show you how to implement the model in PyTorch.

How to do it...

In this recipe, we will define the model, move it to the CUDA device, and get a model summary. Let's get started:

1. First, let's create dumb baselines for the validation dataset.

First, we will get the labels for the validation dataset:

```
# get labels for validation dataset
y_val=[y for _,y in val_ds]
```

Next, we will define a function to calculate the classification accuracy:

```
def accuracy(labels, out):
    return np.sum(out==labels)/float(len(labels))
```

Then, we will calculate a dumb baseline for all-zero predictions:

```
# accuracy all zero predictions
acc_all_zeros=accuracy(y_val,np.zeros_like(y_val))
print("accuracy all zero prediction: %.2f" %acc_all_zeros)

accuracy all zero prediction: 0.60
```

Next, we will calculate a dumb baseline for all-one predictions:

```
# accuracy all ones predictions
acc_all_ones=accuracy(y_val,np.ones_like(y_val))

print("accuracy all one prediction: %.2f" %acc_all_ones)
accuracy all one prediction: 0.40
```

Next, we will calculate a dumb baseline for random predictions:

```
# accuracy random predictions
acc_random=accuracy(y_val,np.random.randint(2,size=len(y_val)))

print("accuracy random prediction: %.2f" %acc_random)

accuracy random prediction: 0.50
```

2. Let's implement a helper function to calculate the output size of a CNN layer.

We will use the following packages:

```
import torch.nn as nn
import numpy as np
```

Then, we'll define the helper function:

```
def findConv2dOutShape(H_in,W_in,conv,pool=2):
    # get conv arguments
    kernel_size=conv.kernel_size
    stride=conv.stride
    padding=conv.padding
    dilation=conv.dilation

    # Ref: https://pytorch.org/docs/stable/nn.html
    H_out=np.floor((H_in+2*padding[0]-
    dilation[0]*(kernel_size[0]-1)-1)/stride[0]+1)
    W_out=np.floor((W_in+2*padding[1]-
    dilation[1]*(kernel_size[1]-1)-1)/stride[1]+1)

    if pool:
        H_out/=pool
        W_out/=pool
    return int(H_out),int(W_out)
```

Next, we will look at the helper function using an example:

```
# example
conv1 = nn.Conv2d(3, 8, kernel_size=3)
h,w=findConv2dOutShape(96,96,conv1)
print(h,w)

47 47
```

3. Next, we will implement the CNN model.

Import the required packages:

```
import torch.nn as nn
import torch.nn.functional as F
```

Define the Net class:

```
class Net(nn.Module):
    def __init__(self, params):
        super(Net, self).__init__()
        C_in,H_in,W_in=params["input_shape"]
        init_f=params["initial_filters"]
        num_fc1=params["num_fc1"]
        num_classes=params["num_classes"]
        self.dropout_rate=params["dropout_rate"]
```

The class continues with the following code:

```
self.conv1 = nn.Conv2d(C_in, init_f, kernel_size=3)
h,w=findConv2dOutShape(H_in,W_in,self.conv1)
self.conv2 = nn.Conv2d(init_f, 2*init_f, kernel_size=3)
h,w=findConv2dOutShape(h,w,self.conv2)
self.conv3 = nn.Conv2d(2*init_f, 4*init_f, kernel_size=3)
h,w=findConv2dOutShape(h,w,self.conv3)

self.conv4 = nn.Conv2d(4*init_f, 8*init_f, kernel_size=3)
h,w=findConv2dOutShape(h,w,self.conv4)
# compute the flatten size
self.num_flatten=h*w*8*init_f
self.fc1 = nn.Linear(self.num_flatten, num_fc1)
self.fc2 = nn.Linear(num_fc1, num_classes)
```

The class continues with the forward function:

```
def forward(self, x):
    x = F.relu(self.conv1(x))
    x = F.max_pool2d(x, 2, 2)
    x = F.relu(self.conv2(x))
    x = F.max_pool2d(x, 2, 2)
    x = F.relu(self.conv3(x))
    x = F.max_pool2d(x, 2, 2)
    x = F.relu(self.conv4(x))
    x = F.max_pool2d(x, 2, 2)
    x = x.view(-1, self.num_flatten)
    x = F.relu(self.fc1(x))
    x=F.dropout(x, self.dropout_rate, training= self.training)
    x = self.fc2(x)
    return F.log_softmax(x, dim=1)
```

4. Then, we will construct an object of the Net class:

```
# dict to define model parameters
params_model={
    "input_shape": (3,96,96),
    "initial_filters": 8,
    "num_fc1": 100,
    "dropout_rate": 0.25,
    "num_classes": 2,
}

# create model
cnn_model = Net(params_model)
```

5. Move the model to a cuda device if one's available:

```
# move model to cuda/gpu device
if torch.cuda.is_available():
    device = torch.device("cuda")
    cnn_model=cnn_model.to(device)
```

6. Print the model:

```
print(cnn_model)

Net (
  (conv1): Conv2d(3, 8, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv3): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (conv4): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=1024, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=2, bias=True)
)
```

7. Verify the model device:

```
print(next(cnn_model.parameters()).device)

cuda:0
```

8. Let's get a summary of the model:

```
from torchsummary import summary
summary(cnn_model, input_size=(3, 96, 96), device=device.type)
```

The model summary is shown in the following code:

```
-----
          Layer (type)           Output Shape        Param #
=====
Conv2d-1            [-1, 8, 94, 94]             224
Conv2d-2            [-1, 16, 45, 45]            1,168
Conv2d-3            [-1, 32, 20, 20]            4,640
Conv2d-4            [-1, 64, 8, 8]             18,496
Linear-5            [-1, 100]                  102,500
Linear-6            [-1, 2]                   202
=====
Total params: 127,230
Trainable params: 127,230
Non-trainable params: 0
-----
Input size (MB): 0.11
```

```
Forward/backward pass size (MB) : 0.92
Params size (MB) : 0.49
Estimated Total Size (MB) : 1.51
-----
```

In the next section, we will explain each step in detail.

How it works...

In *step 1*, we created dumb baselines. We should always establish a baseline before moving forward with complex models. This will help us find out how well we are performing. We can create dumb baselines by creating random predictions, all-zero predictions, or all-one predictions. Let's create simple baselines using these predictions. It is interesting to see that we can get an accuracy of 0.60 with dumb predictions.

In *step 2*, we developed `findConv2DOutShape` to automatically compute the output size of a CNN and pooling layer. The inputs to this function are:

- `H_in`: an integer representing the height of input data
- `W_in`: an integer representing the width of input data
- `conv`: an object of the CNN layer
- `pool`: an integer representing the pooling size and default to 2

The function receives the input size, `H_in`, `W_in`, and `conv` layer and provides the output size, `H_out`, `W_out`. The formula to compute the output size is given in the following link: <https://pytorch.org/docs/stable/nn.html>

In *step 3*, we implemented the CNN model using the `nn.Module` class. In the `__init__` function, we define the layers of the model. We provide the model parameters as a Python dictionary to this function. We recommend using variables for the model parameters and a dictionary to define them outside the class. This way, in the case of parameter changes, we do not have to make changes inside the `Net` class. This will also make future hyperparameter searches easier.

In the `forward` function, we use the defined layers to outline the model. We use `nn.Conv2d` and `nn.Linear` to define the CNN and fully connected layers, respectively. The pooling layers, activations, and dropout layers act like functions and can be defined using `nn.functional`. We do not need to define these layers in the `__init__` function.

Each layer needs specific arguments to be defined.

For most layers, it is important to specify the number of inputs and outputs of the layer. For instance, the first CNN layer has `C_in=3` channels as input and `init_f=8` filters as output, as defined in the following code:

```
self.conv1 = nn.Conv2d(C_in, init_f, kernel_size=3)
```

Similarly, the second CNN layer receives `init_f=8` channels as input and `2*init_f=16` channels as output. Thus, by providing the number of output channels of the previous layer as the number of input channels to the next layer, we can define each layer. However, this becomes tricky when it comes to `nn.Linear` layers. The linear layer accepts a 2D tensor. That is why we need the `view` method in the `forward` function to reshape the 4D tensor into a 2D tensor:

```
# flatten/reshape  
x = x.view(-1, self.num_flatten)
```

How do we get the value of `self.num_flatten`? Well, PyTorch does not automatically calculate this value. It is our duty to find this value and provide it to the linear layer. One approach would be to print (`x.size()`) to get the output shape of each layer.

The other option is using the `findConv2DOutShape` function. The output size of the fourth CNN layer (together with the pooling layer) is `h,w` and there are `8*init_f` output channels. Therefore, after flattening, we get `self.num_flatten=h*w*8*init_f`.

We also added a dropout layer before the output layer to reduce the overfitting problem in deep learning models. Notice that we set the `training = self.training` argument in the `F.dropout` function. The `self.training` parameter is automatically set to `True` during training and `False` at evaluation. This will bypass the dropout layer at the deployment time.



The dropout layer is only applied during training. At deployment, the dropout layer should be deactivated. To do so, make sure to set `training` argument of the dropout layer to `False` at deployment.

The last layer is `F.log_softmax(x, dim=1)` (with two outputs), which is equivalent to $\log(\text{softmax}(x))$. We will explain why we choose this output in the next section when defining the loss function. Keep in mind that if we want to get the output probability value, we need to use the exponential operation.

In *step 4*, we constructed an object of the Net class. The input shape is the shape of the image tensors (3,96,96). We choose `initial_filters=8` filters for the first CNN layer. The number of filters is automatically doubled in the next layers. Also, there are `num_fc1=100` units in the fully connected layer.

In *step 5*, if you have a CUDA device, always move the model to the CUDA device to take advantage of the computation acceleration.



Always use `torch.cuda.is_available()` to avoid getting error in case your computer does not have a CUDA/GPU device.

In *step 6*, we verify that the model was moved to the CUDA device.

In *step 7* and *step 8*, we printed the mode and also got the model summary.

Then, we showed the model summary. The summary shows the output shape and the number of parameters per layer. However, this does not include the functional layers. It is interesting to see that the fully connected layer has the highest number of parameters (102500), which is not efficient. Therefore, state-of-the-art deep learning models eliminate fully connected layers in their design.

See also

We can use pre-trained models and refactor and train them on our data. This is called transfer learning and most of the time leads to better performance. See [Chapter 3, Multi-Class Image Classification](#), for an example of transfer learning using pre-trained models.

Moreover, for details of the arguments of various layers, please visit the following link: <https://pytorch.org/docs/stable/nn.html#>.

Defining the loss function

The standard loss function for classification tasks is cross-entropy loss or logloss. However, when defining the loss function, we need to consider the number of model outputs and their activation functions. For binary classification tasks, we can choose one or two outputs.

The following table shows the corresponding loss functions for different activation functions:

Output Activation	Number of Outputs	Loss Function
None	1	nn.BCEWithLogitsLoss
Sigmoid	1	nn.BCELoss
None	2	nn.CrossEntropyLoss
log_softmax	2	nn.NLLLoss

We recommend using the `log_softmax` function as it is easier to expand to multi-class classification. PyTorch combines the log and softmax operations into one function due to numerical stability and speed.

How to do it...

In this recipe, we will define the loss function and show you how to use it. Let's get started:

1. First, let's define the loss function, as follows:

```
loss_func = nn.NLLLoss(reduction="sum")
```

2. Next, we will use the loss in an example:

```
# fix random seed
torch.manual_seed(0)

n,c=8,2
y = torch.randn(n, c, requires_grad=True)
ls_F = nn.LogSoftmax(dim=1)
y_out=ls_F(y)
print(y_out.shape)

target = torch.randint(c,size=(n,))
print(target.shape)

loss = loss_func(y_out, target)
print(loss.item())

torch.Size([8, 2])
torch.Size([8])
5.266995429992676
```

3. Then, we will compute the gradients of the loss with respect to y :

```
loss.backward()  
print (y.data)  
  
tensor([[-1.1258, -1.1524],  
       [-0.2506, -0.4339],  
       [ 0.8487,  0.6920],  
       [-0.3160, -2.1152],  
       [ 0.3223, -1.2633],  
       [ 0.3500,  0.3081],  
       [ 0.1198,  1.2377],  
       [ 1.1168, -0.2473]])
```

In the next section, we will explain each step in detail.

How it works...

We use `log_softmax` as the output and `nn.NLLLoss` as the negative log-likelihood loss. An important argument in defining the loss function to pay attention to is the reduction, which specifies the reduction to apply to the output. There are three options to choose from: `none`, `sum`, and `mean`. We choose `reduction=sum` so that the output loss will be summed. Since we will process the data in batches, this will return the sum of loss values per batch of data.

In *step 2*, we calculate the loss using an example with $n=8$ samples and $c=2$ classes.

In *step 3*, we compute the gradients for the example in *step 2*. Later, we will use the backward method to compute the gradients of the loss with respect to the model parameters.

See also

Please go to the following link for a list of loss functions that are supported by PyTorch 1.0: <https://pytorch.org/docs/stable/nn.html>.

Defining the optimizer

The `torch.optim` package provides the implementation of common optimizers. The optimizer will hold the current state and will update the parameters based on the computed gradients. For binary classification tasks, SGD and Adam optimizers are used the most.

Another helpful tool in the `torch.optim` package is learning schedules. Learning schedules are useful tools for automatically adjusting the learning rate during training to improve model performance.

How to do it...

In this recipe, we'll learn how to define an optimizer, get the current learning rate, and define a learning scheduler. Let's get started:

1. First, let's define an object of the Adam optimizer with a learning rate of 3e-4:

```
from torch import optim
opt = optim.Adam(cnn_model.parameters(), lr=3e-4)
```

2. We can read the current value of the learning rate using the following function:

```
# get learning rate
def get_lr(opt):
    for param_group in opt.param_groups:
        return param_group['lr']

current_lr=get_lr(opt)
print('current lr={}'.format(current_lr))

current lr=0.0003
```

3. Next, we will define a learning scheduler using the `ReduceLROnPlateau` method:

```
from torch.optim.lr_scheduler import ReduceLROnPlateau

# define learning rate scheduler
lr_scheduler = ReduceLROnPlateau(opt, mode='min', factor=0.5,
                                patience=20, verbose=1)
```

4. Then, we will learn how the learning rate schedule works using the following example:

```
for i in range(100):
    lr_scheduler.step(1)

Epoch    21: reducing learning rate of group 0 to 1.5000e-04.
Epoch    42: reducing learning rate of group 0 to 7.5000e-05.
Epoch    63: reducing learning rate of group 0 to 3.7500e-05.
Epoch    84: reducing learning rate of group 0 to 1.8750e-05.
```

In the next section, we will explain each step in detail.

How it works...

In *step 1*, we defined the Adam optimizer. The important parameters of the optimizer class are the model's parameters and the learning rate. `cnn_model.parameters()` returns an iterator over module parameters that are passed to the optimizer. The learning rate will determine the amount to update by. In PyTorch, we can set a different learning rate per layer, but in this chapter, we'll choose to set one learning rate for all the layers. In *step 2*, we developed a helper function that returns the current value of the learning rate.

In *step 3*, we used the `ReduceLROnPlateau` method from the `torch.optim.lr_scheduler` package. This scheduler reads a metric quantity and if no improvement is seen for a patience number of epochs, the learning rate is reduced by a factor of 0.5. The `mode` argument defines whether the metric quantity is increasing or decreasing during training. For instance, if we monitor the loss value, we set `mode='min'`. If we monitor the accuracy, we should set `mode='max'`.

In *step 4*, we showed you how the learning rate schedule works by passing it a constant value during a loop. As we saw, after every 20 iterations, the scheduler reduces the learning rate by half.

See also

Several methods exist to adjust the learning rate. For a list of supported methods by PyTorch, please visit the following link: <https://pytorch.org/docs/stable/optim.html>.

Training and evaluation of the model

So far, we've created the datasets, built the model, and defined the loss function and optimizer. In this recipe, we'll implement the training and validation scripts. The training and validation scripts can be long and repetitive. For better code readability and to avoid code repetition, we'll build a few helper functions first.

How to do it...

In this recipe, we will show you how to train and evaluate the model using the training and validation dataset. Let's get started:

1. First, let's develop a helper function to count the number of correct predictions per data batch:

```
def metrics_batch(output, target):  
    # get output class  
    pred = output.argmax(dim=1, keepdim=True)  
    # compare output class with target class  
    corrects = pred.eq(target.view_as(pred)).sum().item()  
    return corrects
```

2. Then, we will develop a helper function to compute the loss value per batch of data:

```
def loss_batch(loss_func, output, target, opt=None):  
    loss = loss_func(output, target)  
    with torch.no_grad():  
        metric_b = metrics_batch(output, target)  
    if opt is not None:  
        opt.zero_grad()  
        loss.backward()  
        opt.step()  
    return loss.item(), metric_b
```

3. Next, we develop a helper function to compute the loss value and the performance metric for the entire dataset, also called an epoch.

Define the `loss_epoch` function:

```
def loss_epoch(model, loss_func, dataset_dl, sanity_check=False, opt=None):  
    running_loss=0.0
```

```
running_metric=0.0
len_data=len(dataset_dl.dataset)
```

The function continues with an internal loop over the dataset:

```
for xb, yb in dataset_dl:
    # move batch to device
    xb=xb.to(device)
    yb=yb.to(device)
    # get model output
    output=model(xb)
    # get loss per batch
    loss_b,metric_b=loss_batch(loss_func, output, yb, opt)
```

The loop continues:

```
# update running loss
running_loss+=loss_b
# update running metric
if metric_b is not None:
    running_metric+=metric_b

# break the loop in case of sanity check
if sanity_check is True:
    break
```

The function ends with the following:

```
# average loss value
loss=running_loss/float(len_data)
# average metric value
metric=running_metric/float(len_data)
return loss, metric
```

4. Let's develop the `train_val` function in the following code blocks.

The function is too long, so we will present it in a few steps. Keep in mind that all the following code blocks belong to one function.

First, we extract the model parameters:

```
def train_val(model, params):
    # extract model parameters
    num_epochs=params["num_epochs"]
    loss_func=params["loss_func"]
    opt=params["optimizer"]
    train_dl=params["train_dl"]
    val_dl=params["val_dl"]
```

```
sanity_check=params["sanity_check"]
lr_scheduler=params["lr_scheduler"]
path2weights=params["path2weights"]
```

Then, we define two dictionaries to keep a history of the loss and accuracy values:

```
# history of loss values in each epoch
loss_history={
    "train": [],
    "val": [],
}
# history of metric values in each epoch
metric_history={
    "train": [],
    "val": [],
}
```

Next, we will create a copy of state_dict:

```
# a deep copy of weights for the best performing model
best_model_wts = copy.deepcopy(model.state_dict())
```

Then, we will initialize the best loss to an infinite value:

```
# initialize best loss to a large value
best_loss=float('inf')
```

Next, we will define a loop that will calculate the training loss over an epoch:

```
# main loop
for epoch in range(num_epochs):
    # get current learning rate
    current_lr=get_lr(opt)
    print('Epoch {} / {}, current lr={}'.format(epoch, num_epochs - 1, current_lr))
    # train model on training dataset
    model.train()
    train_loss,
    train_metric=loss_epoch(model, loss_func, train_dl, sanity_check, opt)

    # collect loss and metric for training dataset
    loss_history["train"].append(train_loss)
    metric_history["train"].append(train_metric)
```

Then, we will evaluate the model on the validation dataset:

```
# evaluate model on validation dataset
model.eval()
with torch.no_grad():
```

```

        val_loss,
val_metric=loss_epoch(model, loss_func, val_dl, sanity_check)

# collect loss and metric for validation dataset
loss_history["val"].append(val_loss)
metric_history["val"].append(val_metric)

```

Next, we will store the best weights:

```

# store best model
if val_loss < best_loss:
    best_loss = val_loss
    best_model_wts = copy.deepcopy(model.state_dict())
    # store weights into a local file
    torch.save(model.state_dict(), path2weights)
    print("Copied best model weights!")

```

Then, we will update the learning rate if needed:

```

# learning rate schedule
lr_scheduler.step(val_loss)
if current_lr != get_lr(opt):
    print("Loading best model weights!")
    model.load_state_dict(best_model_wts)

```

Finally, we will print the loss and accuracy values and return the trained model:

```

print("train loss: %.6f, dev loss: %.6f, accuracy: %.2f"
%(train_loss, val_loss, 100*val_metric))
print("-"*10)

# load best model weights
model.load_state_dict(best_model_wts)
return model, loss_history, metric_history

```

5. Let's set the `sanity_check` flag to True and run the code.

Define the objects for the optimization, loss, and learning rate schedule:

```

import copy
loss_func = nn.NLLLoss(reduction="sum")
opt = optim.Adam(cnn_model.parameters(), lr=3e-4)
lr_scheduler = ReduceLROnPlateau(opt, mode='min', factor=0.5,
patience=20, verbose=1)

```

Define the training parameters and call the `train_val` helper function:

```

params_train={
    "num_epochs": 100,

```

```
"optimizer": opt,
"loss_func": loss_func,
"train_dl": train_dl,
"val_dl": val_dl,
"sanity_check": True,
"lr_scheduler": lr_scheduler,
"path2weights": "./models/weights.pt",
}

# train and validate the model
cnn_model, loss_hist, metric_hist = train_val(cnn_model, params_train)
```

This will print out the following output:

```
Epoch 0/99, current lr=0.0003
Copied best model weights!
train loss: 0.000129, dev loss: 0.001024, accuracy: 0.05
-----
Epoch 1/99, current lr=0.0003
Copied best model weights!
train loss: 0.000125, dev loss: 0.001021, accuracy: 0.05
...
...
```

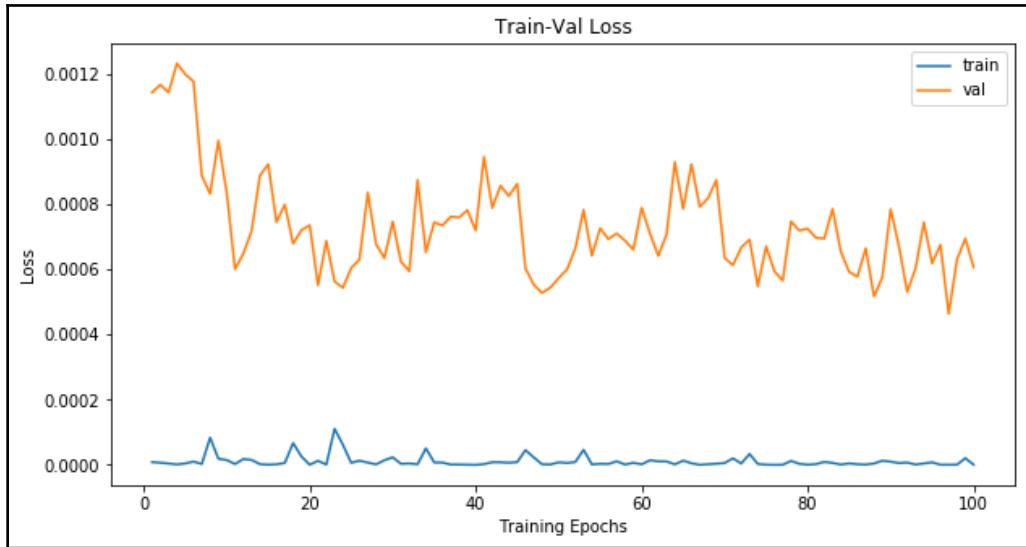
6. Let's plot the training validation's progress using the returned values, that is, `loss_hist` and `metric_hist`:

```
# Train-Validation Progress
num_epochs=params_train["num_epochs"]

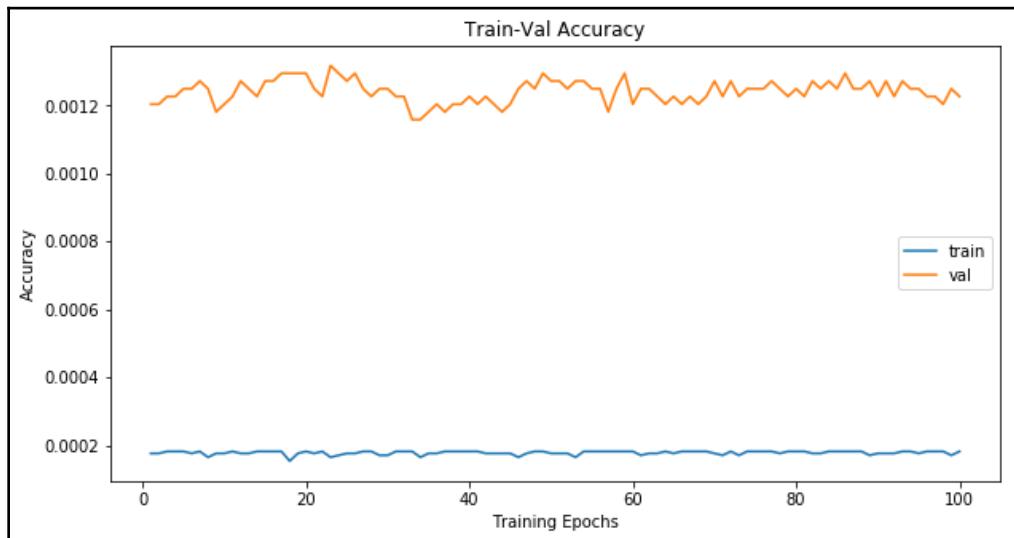
# plot loss progress
plt.title("Train-Val Loss")
plt.plot(range(1,num_epochs+1),loss_hist["train"],label="train")
plt.plot(range(1,num_epochs+1),loss_hist["val"],label="val")
plt.ylabel("Loss")
plt.xlabel("Training Epochs")
plt.legend()
plt.show()

# plot accuracy progress
plt.title("Train-Val Accuracy")
plt.plot(range(1,num_epochs+1),metric_hist["train"],label="train")
plt.plot(range(1,num_epochs+1),metric_hist["val"],label="val")
plt.ylabel("Accuracy")
plt.xlabel("Training Epochs")
plt.legend()
plt.grid()
plt.show()
```

The following screenshot shows the progress of the training and validation losses:



The following screenshot shows the progress of the training and validation metrics:

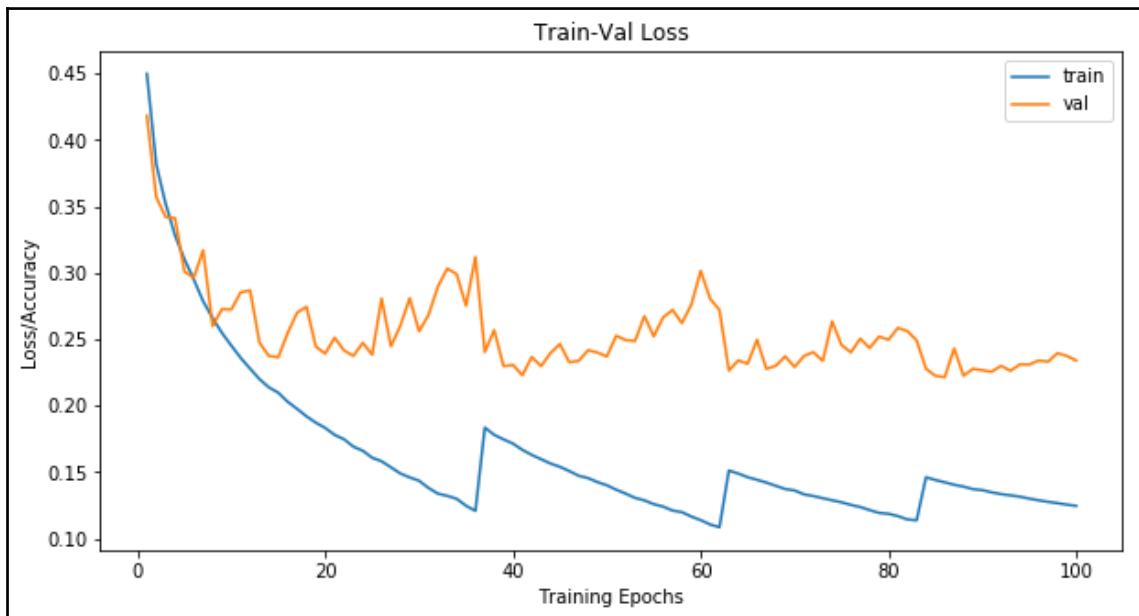


7. By doing this, we've made sure that all the elements are implemented correctly. Let's set the flag to `sanity_check: False` and run the code. The training will start and we will see its progress, as shown in the following code block:

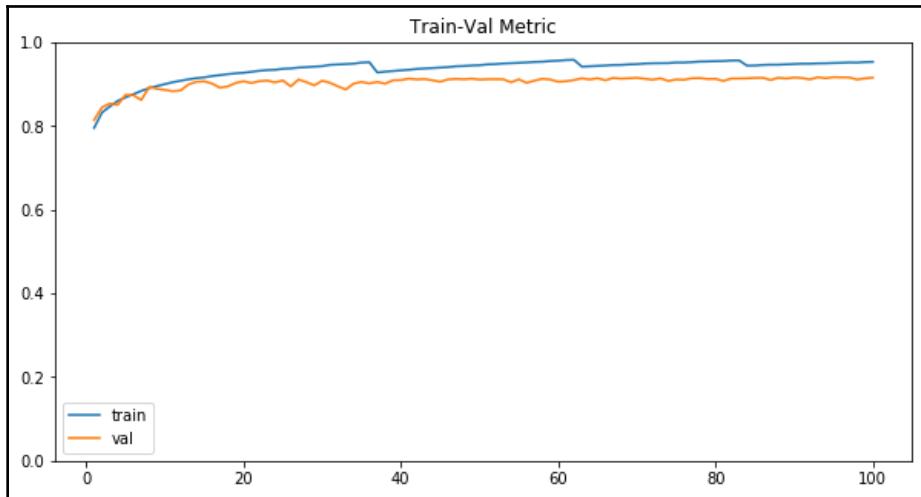
```
Epoch 0/1, current lr=0.0003
Copied best model weights!
train loss: 0.4544370, dev loss: 0.409485, accuracy: 81.75
-----
Epoch 1/1, current lr=0.0003
Copied best model weights!
train loss: 0.381786, dev loss: 0.395184, accuracy: 81.68
-----
...
```

To save space, we've only shown excerpts from the outputs.

The plots of training progress can be seen in the following screenshot:



The plots of metrics can be seen in the following screenshot:



In the next section, we will explain each step in detail.

How it works...

In *step 1*, we counted the number of correct predictions per batch to be used for calculating the model accuracy.

In *step 2*, the `loss_batch` function receives the objects of the loss function and optimizer, the model output, and the ground truth label. We only perform the backpropagation step during training.

In *step 3*, the `loss_epoch` function receives the objects of the model, the loss function, a Dataloader, and the optimizer. We used the Dataloader to fetch batches of data from the dataset. Then, we moved the batch to the CUDA device and got the model's output. We also declared two variables to keep the loss and number of correct predictions after a batch is executed. You can see how the choice of `reduction="sum"` in the definition of the loss function is deployed to accumulate the loss over the entire dataset. The `sanity_check` flag breaks the loop to quickly execute the function in the case of debugging.

In *step 4*, we passed the parameters as a Python dictionary. This will improve the code's readability. We defined two dictionaries to record the loss and metric values during training for visualization purposes.

An important step is to track the model's performance during training to avoid overfitting. To this end, we evaluate the model on the validation dataset after each training epoch. Then, we compared the validation loss with the best validation loss, `val_loss < best_loss`. If the validation loss has improved, we store and make copies of the weights as the best model weights.

The learning rate scheduler monitors the validation loss and reduces the learning rate by a factor of two:

```
# learning rate schedule
lr_scheduler.step(val_loss)
```

Also, every time the learning rate is reduced, we would like to continue training from the best weights:

```
if current_lr != get_lr(opt):
    print("Loading best model weights!")
    model.load_state_dict(best_model_wts)
```

Developing and training deep learning models is time-consuming. The last thing you want is an error popping up somewhere in the middle of training. We recommend performing a sanity check before starting the actual training. We created a flag called `sanity_check` for this purpose. If set to `True`, the training loop breaks after one batch in an epoch. Therefore, we get to see the whole loop in a short amount of time.

In *step 5*, the sanity check should quickly train and evaluate the model for 100 epochs. Everything looks normal. Even the learning rate schedule was activated and reduced the learning rate from 3e-4 to 7.5e-5. The training loss converges to near zero, which makes it clear that we can overfit to one batch. Don't worry about accuracy. This is a sanity check, after all.

In *step 6*, we saw that, at some point, the learning rate schedule will reduce the learning rate after the validation loss does not improve for 20 epochs. When this happens, the model is reloaded with the last best weights. That is why we see sudden jumps in the loss and accuracy curves at these points.

We stored the best model weights (`state_dict`) during training by tracking the model's performance on the validation dataset. The weights are located in the `./models/weights.pt` path. This is useful in case the program is stopped for any reason – at least you have a copy of the best weights.

There's more...

Once you get a baseline performance using your first model, you can start tuning hyperparameters. Hyperparameter tuning is a technique that's used to improve model performance. An effective way of doing this is by using a random search. We defined the hyperparameters as variables so that you can easily play with them. Try different values for the hyperparameters and see how the model's performance is affected. As an example, try increasing the number of CNN filters by setting `initial_filters = 16` and retraining the model.

Deploying the model

We assume that you want to deploy the model for inference in a new script that's separate from the training scripts. In this case, the model and weights do not exist in memory. Therefore, we need to construct an object of the model class and load the weights into the model.

How to do it...

In this recipe, we will define the model, load the weights, and then deploy the model on the validation and test datasets. Let's get started:

1. First, we'll create an object of the `Net` class and load the stored weights into the model:

```
# model parameters
params_model = {
    "input_shape": (3, 96, 96),
    "initial_filters": 8,
    "num_fc1": 100,
    "dropout_rate": 0.25,
    "num_classes": 2,
}

# initialize model
cnn_model = Net(params_model)
```

2. Let's load `state_dict` into the model:

```
# load state_dict into model
path2weights = "./models/weights.pt"
cnn_model.load_state_dict(torch.load(path2weights))
```

3. Set the model in eval mode:

```
# set model in evaluation mode
cnn_model.eval()

Net (
    (conv1): Conv2d(3, 8, kernel_size=(3, 3), stride=(1, 1))
    (conv2): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1))
    (conv3): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
    (conv4): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
    (fc1): Linear(in_features=1024, out_features=100, bias=True)
    (fc2): Linear(in_features=100, out_features=2, bias=True)
)
```

4. Move the model onto a CUDA device if one's available:

```
# move model to cuda/gpu device
if torch.cuda.is_available():
    device = torch.device("cuda")
    cnn_model=cnn_model.to(device)
```

5. Let's develop a helper function to deploy the model on a dataset.

Define the `deploy_model` function:

```
def deploy_model(model,dataset,device,
num_classes=2,sanity_check=False):

    len_data=len(dataset)
    # initialize output tensor on CPU: due to GPU memory limits
    y_out=torch.zeros(len_data,num_classes)
    # initialize ground truth on CPU: due to GPU memory limits
    y_gt=np.zeros((len_data),dtype="uint8")
    # move model to device
    model=model.to(device)
```

The helper function continues with the following code:

```
elapsed_times=[]
with torch.no_grad():
    for i in range(len_data):
        x,y=dataset[i]
        y_gt[i]=y
        start=time.time()
        y_out[i]=model(x.unsqueeze(0).to(device))
        elapsed=time.time()-start
        elapsed_times.append(elapsed)
```

```
        if sanity_check is True:  
            break  
  
        inference_time=np.mean(elapsed_times)*1000  
        print("average inference time per image on %s: %.2f ms "%  
              %(device,inference_time))  
    return y_out.numpy(),y_gt
```

6. Let's use this function to deploy the model on the validation dataset:

```
# deploy model  
y_out,y_gt=deploy_model(cnn_model,val_ds,device=device,sanity_check  
=False)  
print(y_out.shape,y_gt.shape)  
  
average inference time per image on cuda:3: 0.74 ms (44005, 2)  
(44005,)  
(44005, 2) (44005,)
```

7. Let's calculate the accuracy of the model on the validation dataset using the predicted outputs:

```
from sklearn.metrics import accuracy_score  
  
# get predictions  
y_pred = np.argmax(y_out, axis=1)  
print(y_pred.shape,y_gt.shape)  
  
# compute accuracy  
acc=accuracy_score(y_pred,y_gt)  
print("accuracy: %.2f" %acc)  
  
(44005,) (44005,)  
accuracy: 0.91
```

8. Let's also measure the inference time on the CPU device:

```
device_cpu = torch.device("cpu")  
y_out,y_gt=deploy_model(cnn_model,val_ds,device=device_cpu,sanity_c  
heck=False)  
print(y_out.shape,y_gt.shape)  
  
average inference time per image on cpu: 2.21 ms  
(44005, 2) (44005,)
```

In the next section, we will explain each step in detail.

How it works...

In *step 1*, we constructed an object of the model class. Don't forget to copy the `Net` class and the `findConv2dOutShape` function scripts into your new script. When we construct an object of the `Net` class, the weights are randomly initialized.

In *step 2*, we loaded `state_dict`, which contains the model weights, into the model. For deployment, it is necessary to set the model in `eval` mode. This is important since some layers (such as dropout) perform differently in training and deployment modes. If a CUDA device is available, we should move the model onto it. Now, the model is ready to be deployed.

In *step 5*, the helper function returns the model outputs and ground truth labels as NumPy arrays for the dataset. Also, the inference time per image on the CUDA device is estimated to be 0.74 ms. Later, we will see the inference time on the CPU device.

In *step 6*, we verify the stored model by deploying it on the validation dataset. Lots of things could go wrong during the development of deep learning models. We recommend verifying the stored model's performance by deploying the model on a known dataset, for instance, the validation dataset.

In *step 7*, we use the scikit-learn package to calculate the accuracy of our binary classification model. You can install the package using the following code:

```
# install scikit-learn
$ conda install scikit-learn
```

This shows an accuracy of 0.91 on the validation dataset. This provides good verification for our model.

In *step 8*, we want to see the deployment time on the CPU. This is informative if we want to deploy the model on devices without a GPU.

Model inference on test data

Similar to the validation data, we can deploy the model on the test dataset. The labels for the test dataset are not available. Therefore, we will not be able to evaluate the model performance on the test dataset.

Getting ready

Please make a copy of the `sample_submission.csv` file in the `data` folder and rename the copy `test_labels.csv`. We will use the image IDs in `test_labels.csv` in this section.

How to do it...

In this recipe, we will deploy the model on the test dataset and create the submission file for Kaggle submission. Let's get started:

1. First, let's load `test_labels.csv` and print out its head:

```
path2csv="../data/test_labels.csv"  
labels_df=pd.read_csv(path2csv)  
labels_df.head()
```

The following table shows the head of the file:

	ID	Label
0	0b2ea2a822ad23fdb1b5dd26653da899fb2c0d5	0
1	95596b92e5066c5c52466c90b69ff089b39f2737	0
2	248e6738860e2ebcf6258cdc1f32f299e0c76914	0
3	2c35657e312966e9294eac6841726ff3a748feb	0
4	145782eb7caa1c516acbe2eda34d9a3f31c41fd6	0

2. Create a dataset object for the test dataset:

```
histo_test = histoCancerDataset(data_path,  
val_transformer,data_type="test")  
print(len(histo_test))  
57458
```

3. Deploy the model on the test dataset:

```
y_test_out,_=deploy_model(cnn_model,histo_test, device,  
sanity_check=False)  
average inference time per image on cuda:0: 0.74 ms  
  
y_test_pred=np.argmax(y_test_out, axis=1)  
print(y_test_pred.shape)  
(57458,)
```

4. Display a few images and predictions:

```
grid_size=4
rnd_inds=np.random.randint(0,len(histo_test),grid_size)
print("image indices:",rnd_inds)

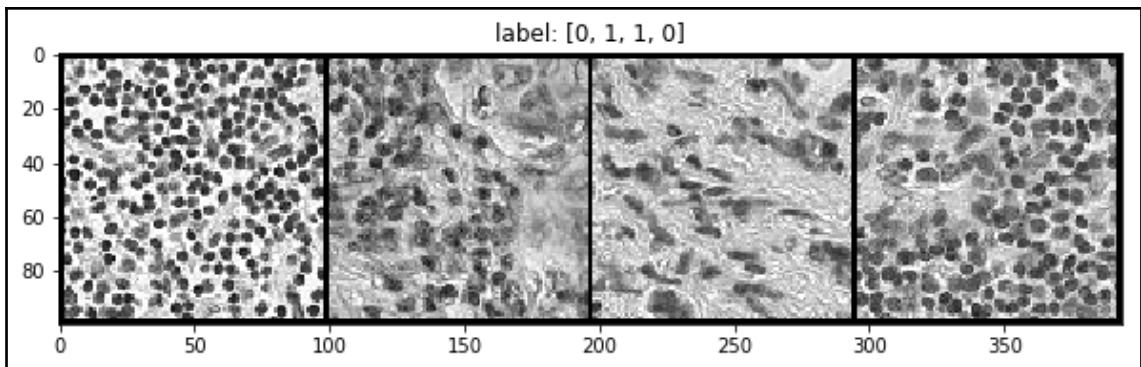
x_grid_test=[histo_test[i][0] for i in range(grid_size)]
y_grid_test=[y_test_pred[i] for i in range(grid_size)]

x_grid_test=utils.make_grid(x_grid_test, nrow=4, padding=2)
print(x_grid_test.shape)

plt.rcParams['figure.figsize'] = (10.0, 5)
show(x_grid_test,y_grid_test)

image indices: [ 2732 43567 42613 52416]
torch.Size([3, 100, 394])
```

Some sample images and predictions from the test dataset can be seen in the following screenshot:



In the next section, we will explain each step in detail.

How it works...

In *step 1*, we loaded the CSV file and print out the head. The labels are all set to zero since we don't know the actual labels. In *step 2*, we created the custom test dataset. The dataset contains 57,458 images. There is no label for these images. Then, we deployed the model on the test dataset and obtained the outputs. In *step 3*, we also displayed a few sample images and the predicted labels.

See also

You can create a submission file from the predictions on the test dataset using the following code block.

First, extract the prediction probabilities from the outputs. Note that we perform the exponential function on the model outputs to convert them into probability values:

```
print(y_test_out.shape)
cancer_preds = np.exp(y_test_out[:, 1])
print(cancer_preds.shape)

(57458, 2)
(57458,)
```

Next, convert the prediction probabilities into a DataFrame and store them as a CSV file:

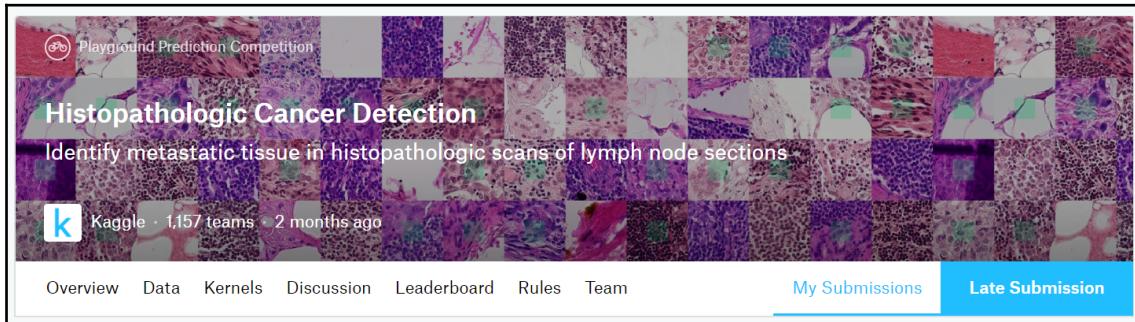
```
path2sampleSub = "./data/" + "sample_submission.csv"
sample_df = pd.read_csv(path2sampleSub)
ids_list = list(sample_df.id)
pred_list = [p for p in cancer_preds]
pred_dic = dict((key[:-4], value) for (key, value) in
zip(histo_test.filenames, pred_list))
pred_list_sub = [pred_dic[id_] for id_ in ids_list]
submission_df = pd.DataFrame({'id':ids_list,'label':pred_list_sub})
if not os.path.exists("./submissions/"):
    os.makedirs("submissions/")
    print("submission folder created!")
path2submission="./submissions/submission.csv"
submission_df.to_csv(path2submission, header=True, index=False)
submission_df.head()
```

The file head can be seen in the following table:

	ID	Label
0	0b2ea2a822ad23fdb1b5dd26653da899fdbd2c0d5	1.646892e-02
1	95596b92e5066c5c52466c90b69ff089b39f2737	5.563063e-03
2	248e6738860e2ebcf6258cdc1f32f299e0c76914	4.128654e-08
3	2c35657e312966e9294eac6841726ff3a748feb	7.437094e-02
4	145782eb7caa1c516acbe2eda34d9a3f31c41fd6	1.990089e-02

You can submit the CSV file to the Histopathologic Cancer Detection competition web page on Kaggle, which can be accessed by going to the following link: <https://www.kaggle.com/c/histopathologic-cancer-detection/submissions>.

Choose **Late Submission** from the menu and upload the file, as shown in the following screenshot:

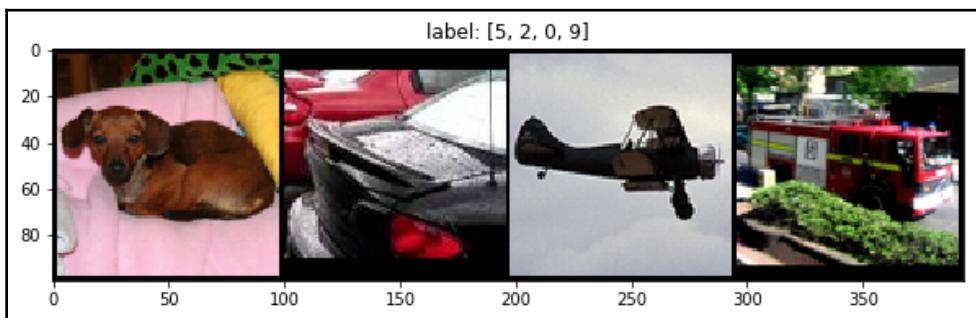


After the file has been uploaded, you will see the leaderboard score for it.

3

Multi-Class Image Classification

Multi-class image classification has been at the center of deep learning's progress. The goal of multi-class image classification is to assign a label to an image from a fixed set of categories. The assumption is that the image contains one dominant object. For instance, the following screenshot shows a few samples from a dataset with 10 categories. We may assign label 5 to dogs, label 2 to cars, label 0 to airplanes, and label 9 to trucks:



In this chapter, we will learn how to create an algorithm to identify 10 categories of objects in the STL-10 dataset. We will use one of the state-of-the-art models pre-trained on the ImageNet dataset and fine-tune it on the STL-10 dataset. The ImageNet dataset, with over 14 million images and 1,000 categories, is one of the famous datasets that helped to push the boundaries in deep learning models. You can learn more about the ImageNet dataset at <http://www.image-net.org/>.

In this chapter, we will cover the following recipes:

- Loading and processing data
- Building the model
- Defining the loss function
- Defining the optimizer

- Training and transfer learning
- Deploying the model

Loading and processing data

We'll use the STL-10 dataset provided in the PyTorch `torchvision` package. There are 10 classes in the dataset:

- Airplane: 0
- Bird: 1
- Car: 2
- Cat: 3
- Deer: 4
- Dog: 5
- Horse: 6
- Monkey: 7
- Ship: 8
- Truck: 9

The images are RGB color with a size of 96*96. The dataset contains 5,000 training images and 8,000 test images. There are 500 and 800 images per class in the training and test datasets, respectively. There are more details about the dataset at <https://cs.stanford.edu/~acoates/stl10/>.

In the following section, we will load the data, display sample images, and create PyTorch dataloaders.

How to do it...

We will start by loading the data and creating the training and validation datasets and dataloaders:

1. Let's load the training data:

```
from torchvision import datasets
import torchvision.transforms as transforms
import os

path2data=". ./data"
```

```
if not os.path.exists(path2data):
    os.mkdir(path2data)
data_transformer = transforms.Compose([transforms.ToTensor()])
train_ds=datasets.STL10(path2data, split='train',
                        download=True, transform=data_transformer)
print(train_ds.data.shape)
```

This snippet will print the following output:

```
(5000, 3, 96, 96)
```

2. Then count the number of images per category in `train_ds`:

```
import collections

y_train=[y for _,y in train_ds]
counter_train=collections.Counter(y_train)
print(counter_train)
```

This snippet will print the following output:

```
Counter({1: 500, 5: 500, 6: 500, 3: 500, 9: 500, 7: 500, 4: 500, 8:
500, 0: 500, 2: 500})
```

3. Let's then load the test dataset and call it `test0_ds`:

```
test0_ds=datasets.STL10(path2data, split='test',
                        download=True, transform=data_transformer)
print(test0_ds.data.shape)
```

This snippet will print the following output:

```
(8000, 3, 96, 96)
```

4. Next, split the indices of `test0_ds` into two groups:

```
from sklearn.model_selection import StratifiedShuffleSplit

sss = StratifiedShuffleSplit(n_splits=1, test_size=0.2,
                             random_state=0)

indices=list(range(len(test0_ds)))
y_test0=[y for _,y in test0_ds]
for test_index, val_index in sss.split(indices, y_test0):
    print("test:", test_index, "val:", val_index)
    print(len(val_index),len(test_index))
```

This snippet will print the following output:

```
test: [2096 4321 2767 ... 3206 3910 2902] val: [6332 6852 1532 ...
5766 4469 1011]
1600 6400
```

5. Then create two datasets from `test0_ds`:

```
from torch.utils.data import Subset

val_ds=Subset(test0_ds,val_index)
test_ds=Subset(test0_ds,test_index)
```

6. Next, count the number of images per class in `val_ds` and `test_ds`:

```
import collections
import numpy as np

y_test=[y for _,y in test_ds]
y_val=[y for _,y in val_ds]

counter_test=collections.Counter(y_test)
counter_val=collections.Counter(y_val)
print(counter_test)
print(counter_val)
```

This snippet will print the following output:

```
Counter({6: 640, 0: 640, 4: 640, 5: 640, 9: 640, 2: 640, 3: 640, 1:
640, 7: 640, 8: 640})
Counter({2: 160, 8: 160, 3: 160, 6: 160, 4: 160, 1: 160, 5: 160, 9:
160, 0: 160, 7: 160})
```

7. Let's show a few sample images from `train_ds`. We will import the required packages:

```
from torchvision import utils
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

np.random.seed(0)
```

Now, we will define the helper function:

```
def show(img,y=None,color=True):
    npimg = img.numpy()
    npimg_tr=np.transpose(npimg, (1,2,0))
```

```
plt.imshow(npimg_tr)
if y is not None:
    plt.title("label: "+str(y))
```

Then pick random samples:

```
grid_size=4
rnd_inds=np.random.randint(0,len(train_ds),grid_size)
print("image indices:",rnd_inds)
```

This snippet will print the following output:

```
image indices: [2732 2607 1653 3264]
```

We will create a grid from the sample images:

```
x_grid=[train_ds[i][0] for i in rnd_inds]
y_grid=[train_ds[i][1] for i in rnd_inds]

x_grid=utils.make_grid(x_grid, nrow=4, padding=1)
print(x_grid.shape)
```

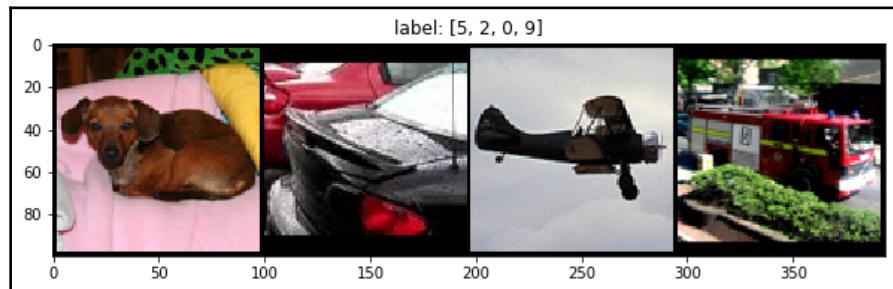
The grid size is printed as follows:

```
torch.Size([3, 100, 394])
```

Then, we will call the helper function to display the grid:

```
# call helper function
plt.figure(figsize=(10,10))
show(x_grid,y_grid)
```

The following screenshot shows sample images and corresponding labels from the training dataset:



8. Let's show sample images from val_ds:

```
np.random.seed(0)

grid_size=4
rnd_inds=np.random.randint(0,len(val_ds),grid_size)
print("image indices:",rnd_inds)

x_grid=[val_ds[i][0] for i in rnd_inds]
y_grid=[val_ds[i][1] for i in rnd_inds]

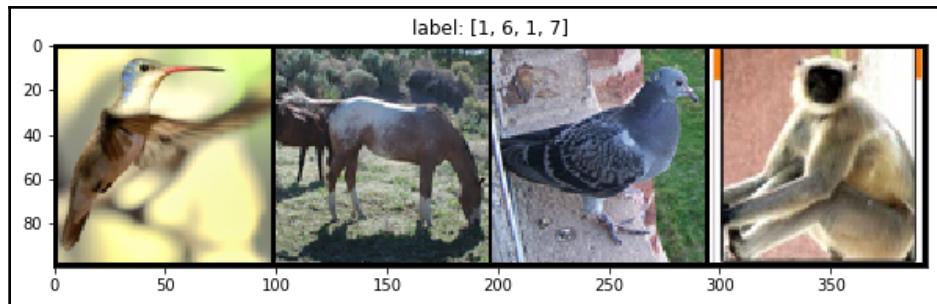
x_grid=utils.make_grid(x_grid, nrow=4, padding=2)
print(x_grid.shape)

plt.figure(figsize=(10,10))
show(x_grid,y_grid)
```

This snippet will print the following output:

```
image indices: [ 684  559 1216  835]
torch.Size([3, 100, 394])
```

The following screenshot depicts sample images and corresponding labels from val_ds:



9. Let's calculate the mean and standard deviation of train_ds:

```
import numpy as np

meanRGB=[np.mean(x.numpy(),axis=(1,2)) for x,_ in train_ds]
stdRGB=[np.std(x.numpy(),axis=(1,2)) for x,_ in train_ds]

meanR=np.mean([m[0] for m in meanRGB])
meanG=np.mean([m[1] for m in meanRGB])
meanB=np.mean([m[2] for m in meanRGB])
```

```
stdR=np.mean([s[0] for s in stdRGB])
stdG=np.mean([s[1] for s in stdRGB])
stdB=np.mean([s[2] for s in stdRGB])

print(meanR,meanG,meanB)
print(stdR,stdG,stdB)
```

This snippet will print the following output:

```
0.4467106 0.43980986 0.40664646
0.22414584 0.22148906 0.22389975
```

10. Let's define the image transformations for `train_ds` and `test0_ds`:

```
train_transformer = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.5),
    transforms.ToTensor(),
    transforms.Normalize([meanR, meanG, meanB], [stdR, stdG,
stdB]))]

test0_transformer = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([meanR, meanG, meanB], [stdR, stdG,
stdB]),
])
```

11. Update the `transform` functions of `train_ds` and `test0_ds`:

```
train_ds.transform=train_transformer
test0_ds.transform=test0_transformer
```

12. Next, we will display the transformed sample images from `train_ds`:

```
import torch
np.random.seed(0)
torch.manual_seed(0)

grid_size=4
rnd_inds=np.random.randint(0,len(train_ds),grid_size)
print("image indices:",rnd_inds)

x_grid=[train_ds[i][0] for i in rnd_inds]
y_grid=[train_ds[i][1] for i in rnd_inds]

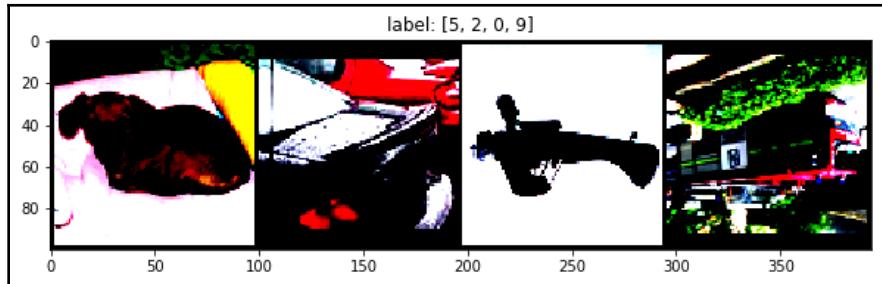
x_grid=utils.make_grid(x_grid, nrow=4, padding=2)
print(x_grid.shape)
```

```
plt.figure(figsize=(10,10))
show(x_grid,y_grid)
```

This snippet will print the following output:

```
image indices: [2732 2607 1653 3264]
torch.Size([3, 100, 394])
```

The sample transformed images are shown in the following screenshot:



13. Let's create dataloaders from `train_ds` and `val_ds`:

```
from torch.utils.data import DataLoader

train_dl = DataLoader(train_ds, batch_size=32, shuffle=True)
val_dl = DataLoader(val_ds, batch_size=64, shuffle=False)
```

14. Then get a batch of data from `train_dl`:

```
for x, y in train_dl:
    print(x.shape)
    print(y.shape)
    break
```

This snippet will print the following output:

```
torch.Size([32, 3, 96, 96])
torch.Size([32])
```

15. Next, get a batch of data from `val_dl`:

```
# extract a batch from validation data
for x, y in val_dl:
    print(x.shape)
    print(y.shape)
    break
```

This snippet will print the following output:

```
torch.Size([64, 3, 96, 96])
torch.Size([64])
```

In the next section, we will explain each step in detail.

How it works...

In *step 1*, we loaded the training dataset from `torchvision.datasets`. As seen, there are 5,000 images of size $3 \times 96 \times 96$ in the training dataset. The initial `transform` function converts PIL images to tensors and normalizes pixel values in the range $[0, 1]$. Make sure that `Download = True` the first time you run the code. It will automatically download the dataset and store it in the data folder. After downloading once, you can set `Download=False`. In *step 2*, we counted the number of images per class. As seen, there is an equal number of images in each class (a balanced dataset).

In *step 3*, we loaded the original test dataset and called it `test0_ds`. As seen, there are 8,000 images in `test0_ds`. Since there is no official validation dataset, we split `test0_ds` indices into two groups, `val_index`, and `test_index`. In *step 4* we used `StratifiedShuffleSplit` from `sklearn.model_selection`. This function returns stratified randomized indices. In *step 5*, we used `val_index` and `test_index` to create two datasets: `val_ds` and `test_ds`. There are 1,600 images in `val_ds` and 6,400 remaining images in `test_ds`. We will use `val_ds` for the evaluation of the model during training.

Since we used the stratified randomized indices, we can see in *step 6* that both `val_ds` and `test_ds` are balanced. In *step 7* and *step 8*, we displayed sample images from `train_ds` and `val_ds`. We fixed the random seed using `np.random.seed(0)` so that the random indices are reproduced in different runs. If you want to see different random images in each run, comment out the line. In *step 9*, we calculated the mean and standard deviation of pixel values in `train_ds`. We will use these values for the normalization of data.

In *step 10*, we defined image transformations. For `train_ds`, we added `RandomHorizontalFlip` and `RandomVerticalFlip` to augment the training dataset. In addition, we applied zero-mean unit-variance normalization using `transforms.Normalize`. The arguments to this function are the mean and standard deviation per channel calculated in *step 9*. For `test0_ds`, we only added the normalization function since we do not require data augmentation for the validation and test datasets.

In *step 11*, we updated the `transform` functions of `train_ds` and `test0_ds`. Note that when we update the `test0_ds.transform` function, both `val_ds`, and `test_ds` will be updated since they are subsets of `test0_ds`. In *step 12*, we displayed sample images from `train_ds` with updated transformation. Note that, due to normalization, the pixel values of transformed images are clipped when displayed. As seen, three images are flipped horizontally or vertically.

In *step 13*, we created the PyTorch dataloaders `train_dl` and `val_dl` to be able to automatically fetch data batches from each dataset. In *step 14* and *step 15*, we extracted a batch of data using the dataloaders. As seen, the extracted data size depends on the batch size.

There's more...

There are other built-in datasets in the `torchvision` package that can be similarly loaded for multi-class image classification. For example, you can load `FashionMNIST`, consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes.

Try the following code to load the `FashionMNIST` dataset:

```
from torchvision import datasets
fashion_train=datasets.FashionMNIST(path2data, train=True, download=True)
```

Do you want to play with more datasets? See the next section to explore more.

See also

Click on the following link for a list of all built-in datasets in `torchvision.datasets`: <https://pytorch.org/docs/0.4.0/torchvision/datasets.html>.

Building the model

In this recipe, we will build a model for our multi-class classification task. Instead of building a custom model, we'll use the torchvision models. The `torchvision` package provides the implementation of multiple state-of-the-art deep learning models for image classification. These include AlexNet, VGG, ResNet, SqueezeNet, DenseNet, Inception, GoogleNet, and ShuffleNet. These models were trained on the ImageNet dataset with over 14 million images from 1,000 classes. We can either use the architectures with randomly initialized weights or the pre-trained weights. In this recipe, we will try both cases and see the difference.

How to do it...

We will load a pre-trained model and visualize the convolutional layers in the first layer:

1. Let's import the `resnet18` model with random weights from `torchvision.models`:

```
from torchvision import models
import torch

model_resnet18 = models.resnet18(pretrained=False)
```

2. Let's print the model:

```
print(model_resnet18)
```

This snippet will print the model:

```
ResNet (
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2),
  padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
  track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1,
  dilation=1, ceil_mode=False)
  ...
)
```

You will see the full model printed. For brevity, we only showed the beginning of the model in the preceding snippet.

3. Let's change the output layer to 10 classes:

```
from torch import nn

num_classes=10
num_ftrs = model_resnet18.fc.in_features
model_resnet18.fc = nn.Linear(num_ftrs, num_classes)

device = torch.device("cuda:0")
model_resnet18.to(device)
```

4. Let's get the model summary:

```
from torchsummary import summary
summary(model_resnet18, input_size=(3,224,224), device=device.type)
```

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 112, 112]	9,408
BatchNorm2d-2	[-1, 64, 112, 112]	128
ReLU-3	[-1, 64, 112, 112]	0
MaxPool2d-4	[-1, 64, 56, 56]	0
Conv2d-5	[-1, 64, 56, 56]	36,864
BatchNorm2d-6	[-1, 64, 56, 56]	128
ReLU-7	[-1, 64, 56, 56]	0
Conv2d-8	[-1, 64, 56, 56]	36,864
BatchNorm2d-9	[-1, 64, 56, 56]	128

The model summary continues as follows:

ReLU-10	[-1, 64, 56, 56]	0
BasicBlock-11	[-1, 64, 56, 56]	0
Conv2d-12	[-1, 64, 56, 56]	36,864
BatchNorm2d-13	[-1, 64, 56, 56]	128
ReLU-14	[-1, 64, 56, 56]	0
Conv2d-15	[-1, 64, 56, 56]	36,864
BatchNorm2d-16	[-1, 64, 56, 56]	128
ReLU-17	[-1, 64, 56, 56]	0
BasicBlock-18	[-1, 64, 56, 56]	0
Conv2d-19	[-1, 128, 28, 28]	73,728
BatchNorm2d-20	[-1, 128, 28, 28]	256
ReLU-21	[-1, 128, 28, 28]	0
Conv2d-22	[-1, 128, 28, 28]	147,456
BatchNorm2d-23	[-1, 128, 28, 28]	256
Conv2d-24	[-1, 128, 28, 28]	8,192

It continues further:

BatchNorm2d-25	$[-1, 128, 28, 28]$	256
ReLU-26	$[-1, 128, 28, 28]$	0
BasicBlock-27	$[-1, 128, 28, 28]$	0
Conv2d-28	$[-1, 128, 28, 28]$	147,456
BatchNorm2d-29	$[-1, 128, 28, 28]$	256
ReLU-30	$[-1, 128, 28, 28]$	0
Conv2d-31	$[-1, 128, 28, 28]$	147,456
BatchNorm2d-32	$[-1, 128, 28, 28]$	256
ReLU-33	$[-1, 128, 28, 28]$	0
BasicBlock-34	$[-1, 128, 28, 28]$	0
Conv2d-35	$[-1, 256, 14, 14]$	294,912
BatchNorm2d-36	$[-1, 256, 14, 14]$	512
ReLU-37	$[-1, 256, 14, 14]$	0

It continues still further here:

Conv2d-38	$[-1, 256, 14, 14]$	589,824
BatchNorm2d-39	$[-1, 256, 14, 14]$	512
Conv2d-40	$[-1, 256, 14, 14]$	32,768
BatchNorm2d-41	$[-1, 256, 14, 14]$	512
ReLU-42	$[-1, 256, 14, 14]$	0
BasicBlock-43	$[-1, 256, 14, 14]$	0
Conv2d-44	$[-1, 256, 14, 14]$	589,824
BatchNorm2d-45	$[-1, 256, 14, 14]$	512
ReLU-46	$[-1, 256, 14, 14]$	0
Conv2d-47	$[-1, 256, 14, 14]$	589,824
BatchNorm2d-48	$[-1, 256, 14, 14]$	512
ReLU-49	$[-1, 256, 14, 14]$	0
BasicBlock-50	$[-1, 256, 14, 14]$	0

And here:

Conv2d-51	$[-1, 512, 7, 7]$	1,179,648
BatchNorm2d-52	$[-1, 512, 7, 7]$	1,024
ReLU-53	$[-1, 512, 7, 7]$	0
Conv2d-54	$[-1, 512, 7, 7]$	2,359,296
BatchNorm2d-55	$[-1, 512, 7, 7]$	1,024
Conv2d-56	$[-1, 512, 7, 7]$	131,072
BatchNorm2d-57	$[-1, 512, 7, 7]$	1,024
ReLU-58	$[-1, 512, 7, 7]$	0
BasicBlock-59	$[-1, 512, 7, 7]$	0
Conv2d-60	$[-1, 512, 7, 7]$	2,359,296
BatchNorm2d-61	$[-1, 512, 7, 7]$	1,024
ReLU-62	$[-1, 512, 7, 7]$	0
Conv2d-63	$[-1, 512, 7, 7]$	2,359,296

The model summary finally ends here:

```

BatchNorm2d-64           [-1, 512, 7, 7]      1,024
    ReLU-65                [-1, 512, 7, 7]      0
    BasicBlock-66          [-1, 512, 7, 7]      0
AdaptiveAvgPool2d-67     [-1, 512, 1, 1]      0
    Linear-68              [-1, 10]            5,130
=====
Total params: 11,181,642
Trainable params: 11,181,642
Non-trainable params: 0
-----
Input size (MB): 0.57
Forward/backward pass size (MB): 62.79
Params size (MB): 42.65
Estimated Total Size (MB): 106.01
-----
```

5. Let's visualize the filters of the first CNN layer:

Let's get the weights of the first layer:

```

for w in model_resnet18.parameters():
    w=w.data.cpu()
    print(w.shape)
    break
```

Then, normalize the weights:

```

min_w=torch.min(w)
w1 = (-1/(2*min_w))*w + 0.5
print(torch.min(w1).item(),torch.max(w1).item())
```

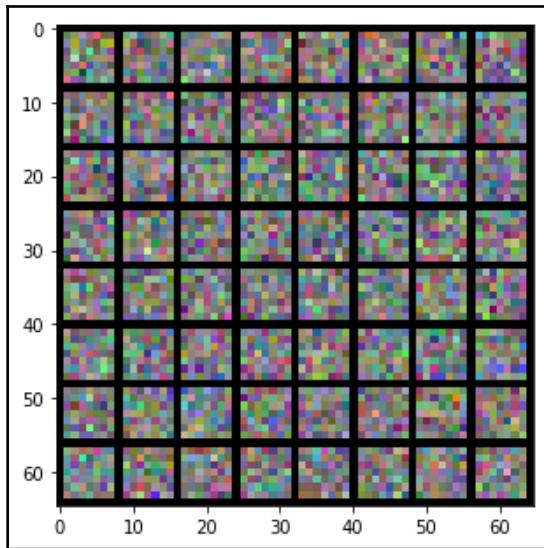
Next, make a grid and display it:

```

grid_size=len(w1)
x_grid=[w1[i] for i in range(grid_size)]
x_grid=utils.make_grid(x_grid, nrow=8, padding=1)
print(x_grid.shape)

plt.figure(figsize=(10,10))
show(x_grid)
```

The filters are shown in the following screenshot:



6. Let's load `resnet18` with the pre-trained weights:

```
from torchvision import models
import torch

resnet18_pretrained = models.resnet18(pretrained=True)

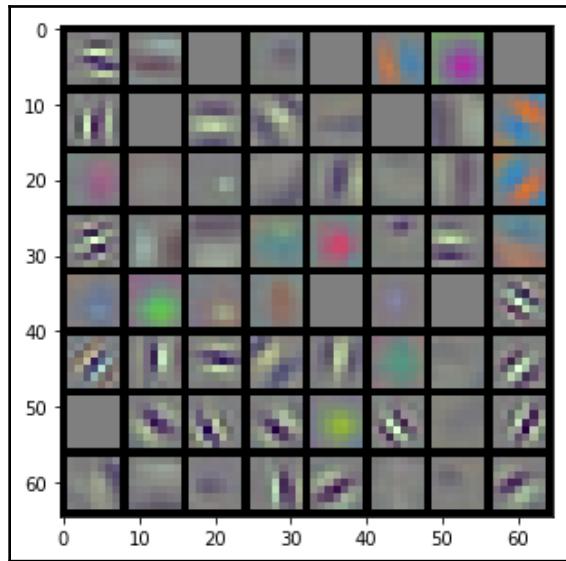
num_classes=10
num_ftrs = resnet18_pretrained.fc.in_features
resnet18_pretrained.fc = nn.Linear(num_ftrs, num_classes)

device = torch.device("cuda:0")
resnet18_pretrained.to(device)
```

7. Let's visualize the filters of the first CNN layer using the snippet from *step 5*:

```
# use the code snippet in step 5
```

The filters are shown in the following screenshot:



In the next section, we will explain each step in detail.

How it works...

In *step 1*, we loaded the `resnet18` model. This will automatically download the model and store it locally for future use. An important argument when loading a model is `pretrained`. If set to `False`, the model weights will be initialized randomly.

In *step 2*, we printed the model. As seen, the last layer is a linear layer with 1,000 outputs. The `resnet18` model was developed for the `ImageNet` dataset with 1,000 classes. Therefore, we change the last layer to have `num_classes = 10` outputs for our classification task in *step 3*. This change can be seen in *step 4* when we get the model summary. Even though the original image sizes are $96*96$, we need to resize them to $224*224$, the same size that the `resnet18` model was trained at.

In *step 5*, we visualized the convolutional filters of the first layer. Since filters are randomly initialized, they show random patterns without any structure. Later, we will see the same filters for the pre-trained model.

In *step 6*, we loaded the `resnet18` model with `pretrained = True`. This will load the model with the pre-trained weights on the ImageNet dataset. In *step 7*, we displayed the filters of the first layer. As seen, the filters represent some patterns and structures such as edges.

There's more...

The `torchvision.models` package provides the definitions of more models. Try to load other pre-trained models and change the last layer to match our dataset with 10 classes. For instance, you can load `vgg19` using the following code:

```
num_classes=10
vgg19 = models.vgg19(pretrained=True)
# change the last layer
vgg19.classifier[6] = nn.Linear(4096,num_classes)
```

See also

You can find the list of models at <https://pytorch.org/docs/stable/torchvision/models.html>. Also, you can check out the tutorial at the following link to see examples of loading the models: https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html.

Defining the loss function

The goal of defining a loss function is to optimize the model toward a pre-defined metric. The standard loss function for classification tasks is cross-entropy loss or log loss. However, in defining the loss function, we need to consider the number of model outputs and their activation functions. For multi-class classification tasks, the number of outputs is set to the number of classes. The output activation function then determines the loss function.

The following table shows the corresponding loss function for different activation functions:

Output activation	Number of outputs	Loss function
None	num_classes	nn.CrossEntropyLoss
log_Softmax	num_classes	nn.NLLLoss

The `resnet18` model defined in the *Building the model* section has linear outputs with no activation function. Therefore, we choose `nn.CrossEntropyLoss` as the loss function. This loss function combines `nn.LogSoftmax()` and `nn.NLLLoss()` in one class.

How to do it...

We will define the loss function and test it with random values:

1. Define the loss function as the following:

```
loss_func = nn.CrossEntropyLoss(reduction="sum")
```

2. Let's see the loss in an example:

```
torch.manual_seed(0)

n, c=4, 5
y = torch.randn(n, c, requires_grad=True)
print(y.shape)

loss_func = nn.CrossEntropyLoss(reduction="sum")
target = torch.randint(c, size=(n,))
print(target.shape)

loss = loss_func(y, target)
print(loss.item())
```

This snippet will print the following output:

```
torch.Size([4, 5])
torch.Size([4])
7.312585830688477
```

3. Let's compute the gradients of loss with respect to y :

```
loss.backward()  
print (y.data)  
  
tensor([[-1.1258, -1.1524, -0.2506, -0.4339,  0.5988],  
       [-1.5551, -0.3414,  1.8530,  0.4681, -0.1577],  
       [ 1.4437,  0.2660,  1.3894,  1.5863,  0.9463],  
       [-0.8437,  0.9318,  1.2590,  2.0050,  0.0537]])
```

In the next section, we will explain these steps in detail.

How it works...

In *step 1*, we defined the loss function. The `resnet18` model uses linear outputs. Thus, we use `nn.CrossEntropyLoss` as the loss function. An important argument in defining the loss function to pay attention to is `reduction`, which specifies the reduction to apply to the output. There are three options to choose from: `none`, `sum`, and `mean`. We choose `reduction="sum"` so the output loss will be summed. Since we will process the data in batches, this will return the sum of loss values per batch of data.

In *step 2*, we calculated the loss using an example with $n=4$ samples and $c=5$ classes.

In *step 3*, we computed the gradients for the example in *step 2*. Later, we will use the `backward` method to compute the gradients of loss with respect to the model parameters.

See also

Visit the following link for a list of the loss functions supported by PyTorch 1.0: <https://pytorch.org/docs/stable/nn.html>.

Defining the optimizer

The `torch.optim` package provides the implementation of common optimizers. The optimizer will hold the current state and will update the parameters based on the computed gradients. For classification tasks, **Stochastic Gradient Descent (SGD)** and Adam optimizer are very common to use. The choice of an optimizer for your model is considered a hyperparameter. You usually need to try multiple optimizers to find the best performing one. Adam optimizer outperforms the SGD in terms of speed and accuracy most often, so we will choose Adam optimizer here. However, you can try a different optimizer and see for yourself.

Other nice tools in the `torch.optim` package are learning schedules. Learning schedules are useful tools to automatically adjust the learning rate during training to improve model performance.

In this recipe, we will learn how to define an optimizer, get the current learning rate, and define a learning scheduler.

How to do it...

We will define the optimizer and the learning rate schedule:

1. Define an `Adam` optimizer object with a learning rate of `1e-4`:

```
from torch import optim
opt = optim.Adam(model_resnet18.parameters(), lr=1e-4)
```

2. We can read the current value of the learning rate using the following function:

```
def get_lr(opt):
    for param_group in opt.param_groups:
        return param_group['lr']

current_lr=get_lr(opt)
print('current lr={}'.format(current_lr))
```

This snippet will print the following output:

```
current lr=0.0001
```

3. Define a learning scheduler using the `CosineAnnealingLR` method:

```
from torch.optim.lr_scheduler import CosineAnnealingLR  
  
lr_scheduler = CosineAnnealingLR(opt, T_max=2, eta_min=1e-5)
```

4. Let's see how the learning rate schedule works using the following example:

```
for i in range(10):  
    lr_scheduler.step()  
    print("epoch %s, lr: %.1e" %(i, get_lr(opt)))  
  
epoch 0, lr: 1.0e-04  
epoch 1, lr: 5.5e-05  
epoch 2, lr: 1.0e-05  
epoch 3, lr: 5.5e-05  
epoch 4, lr: 1.0e-04  
epoch 5, lr: 5.5e-05  
epoch 6, lr: 1.0e-05  
epoch 7, lr: 5.5e-05  
epoch 8, lr: 1.0e-04  
epoch 9, lr: 5.5e-05
```

In the next section, we will explain each step in detail.

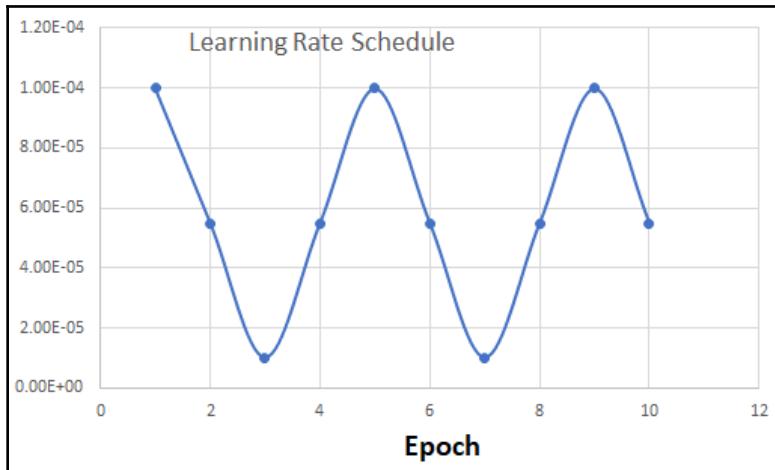
How it works...

In *step 1*, we defined Adam optimizer. As mentioned before, Adam optimizer outperforms other optimizers for classification tasks most of the time. However, this should not be considered a rule. Consider choosing the optimizer as a hyperparameter and try a few different optimizers to see which one performs better. The important parameters of the optimizer class are the model parameters and the learning rate.

`model_resnet18.parameters()` returns an iterator over module parameters that are passed to the optimizer. The learning rate will determine the number of updates. We set one learning rate for all the layers. In *step 2*, we developed a helper function that returns the current value of the learning rate.

In *step 3*, we used the `CosineAnnealingLR` method from the `torch.optim.lr_scheduler` package. This scheduler adjusts the learning rate based on a cosine annealing schedule. The learning rate starts from the set value in the optimizer $lr=1e-4$. Then, it gradually decreases toward $eta_min=1e-5$ and returns back to the original set value in $2*T_max=4$ iterations.

The following screenshot depicts a plot of the learning rate values over 10 epochs:



Later, we will see how we can call this learning-rate schedule during training.

See also

Several methods exist to adjust the learning rate. For a list of supported methods by PyTorch, please visit the following link: <https://pytorch.org/docs/stable/optim.html>.

Training and transfer learning

By now, we have created the datasets and defined the model, loss function, and optimizer. In this recipe, we will implement the training and validation scripts. We'll first train the model with randomly initialized weights. Then, we'll train the model with the pre-trained weights. This technique is also called transfer learning. In transfer learning, we try to use the knowledge (weights) learned from one problem in other similar problems. The training and validation scripts could be long and repetitive. For better code readability and to avoid code repetition, we'll first build a few helper functions.

How to do it...

We will train the model from scratch and using transfer learning:

1. First, develop a helper function to count the number of correct predictions per data batch:

```
def metrics_batch(output, target):  
    pred = output.argmax(dim=1, keepdim=True)  
    corrects=pred.eq(target.view_as(pred)).sum().item()  
    return corrects
```

2. We then develop a helper function to compute the loss value per batch of data:

```
def loss_batch(loss_func, output, target, opt=None):  
    loss = loss_func(output, target)  
    metric_b = metrics_batch(output,target)  
    if opt is not None:  
        opt.zero_grad()  
        loss.backward()  
        opt.step()  
  
    return loss.item(), metric_b
```

3. Next, we develop a helper function to compute the loss value and the performance metric for the entire dataset or an epoch.

We start the helper function by initializing variables:

```
def  
loss_epoch(model, loss_func, dataset_dl, sanity_check=False, opt=None):  
    running_loss=0.0  
    running_metric=0.0  
    len_data=len(dataset_dl.dataset)
```

The helper function continues with an internal loop:

```
for xb, yb in dataset_dl:  
    xb=xb.to(device)  
    yb=yb.to(device)  
    output=model(xb)  
    loss_b,metric_b=loss_batch(loss_func, output, yb, opt)  
    running_loss+=loss_b
```

The function ends with the following:

```

if metric_b is not None:
    running_metric+=metric_b

if sanity_check is True:
    break

loss=running_loss/float(len_data)
metric=running_metric/float(len_data)
return loss, metric

```

4. Let's develop the `train_val` function in the following code block.

The helper function starts by extracting the parameters:

```

def train_val(model, params):
    num_epochs=params["num_epochs"]
    loss_func=params["loss_func"]
    opt=params["optimizer"]
    train_dl=params["train_dl"]
    val_dl=params["val_dl"]
    sanity_check=params["sanity_check"]
    lr_scheduler=params["lr_scheduler"]
    path2weights=params["path2weights"]

```

The helper function continues with two dictionaries to store the loss and metric:

```

loss_history={
    "train": [],
    "val": []
}
metric_history={
    "train": [],
    "val": []
}

```

Then we will define variables to store the best model parameters:

```

best_model_wts = copy.deepcopy(model.state_dict())
best_loss=float('inf')

```

The helper function continues with an internal loop:

```

for epoch in range(num_epochs):
    current_lr=get_lr(opt)
    print('Epoch {}/{} , current lr={}'.format(epoch, num_epochs - 1, current_lr))
    model.train()

```

```

        train_loss,
train_metric=loss_epoch(model, loss_func, train_dl, sanity_check, opt)

        loss_history["train"].append(train_loss)
metric_history["train"].append(train_metric)

```

The loop continues by evaluating the model:

```

model.eval()
with torch.no_grad():
    val_loss,
val_metric=loss_epoch(model, loss_func, val_dl, sanity_check)
    loss_history["val"].append(val_loss)
metric_history["val"].append(val_metric)

```

The loop continues with the best model parameters:

```

if val_loss < best_loss:
    best_loss = val_loss
    best_model_wts = copy.deepcopy(model.state_dict())
    torch.save(model.state_dict(), path2weights)
    print("Copied best model weights!")

```

The loop ends with the learning rate schedule and by printing the loss and metric values:

```

lr_scheduler.step()

print("train loss: %.6f, dev loss: %.6f, accuracy: %.2f"
%(train_loss, val_loss, 100*val_metric))
print("-"*10)

```

Finally, the helper function ends by returning the best model:

```

model.load_state_dict(best_model_wts)
return model, loss_history, metric_history

```

5. Let's train the model by calling the `train_val` function.

We will redefine the loss, optimizer, and learning rate schedule:

```

import copy

loss_func = nn.CrossEntropyLoss(reduction="sum")
opt = optim.Adam(model_resnet18.parameters(), lr=1e-4)
lr_scheduler = CosineAnnealingLR(opt, T_max=5, eta_min=1e-6)

```

We will set the training parameters and call the `train_val` function:

```
os.makedirs("./models", exist_ok=True)

params_train={
    "num_epochs": 100,
    "optimizer": opt,
    "loss_func": loss_func,
    "train_dl": train_dl,
    "val_dl": val_dl,
    "sanity_check": False,
    "lr_scheduler": lr_scheduler,
    "path2weights": "./models/resnet18.pt",
}

model_resnet18, loss_hist, metric_hist = train_val(model_resnet18, params_train)
```

The training will start and you should see its progress:

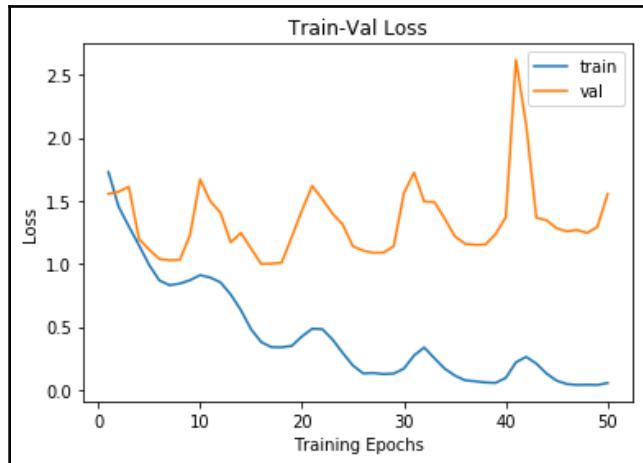
```
Epoch 0/2, current lr=0.0001
Copied best model weights!
train loss: 1.772097, dev loss: 1.615586, accuracy: 38.06
-----
Epoch 1/2, current lr=0.0001
Copied best model weights!
train loss: 1.429164, dev loss: 1.522995, accuracy: 44.25
-----
Epoch 2/2, current lr=9.05463412215599e-05
Copied best model weights!
train loss: 1.258205, dev loss: 1.442116, accuracy: 48.25
-----
```

6. Next, we will plot the progress of the loss values throughout the training:

```
num_epochs=params_train["num_epochs"]

plt.title("Train-Val Loss")
plt.plot(range(1,num_epochs+1),loss_hist["train"],label="train")
plt.plot(range(1,num_epochs+1),loss_hist["val"],label="val")
plt.ylabel("Loss")
plt.xlabel("Training Epochs")
plt.legend()
plt.show()
```

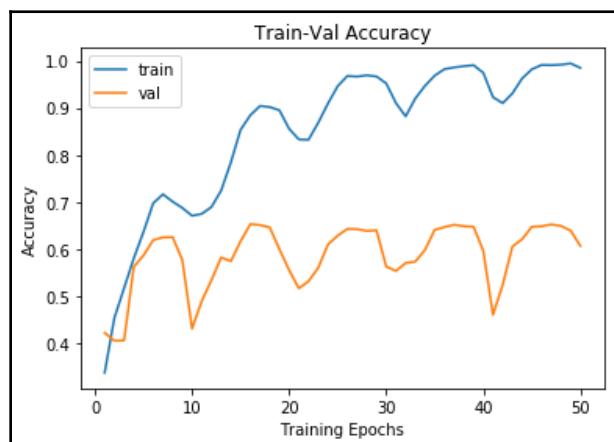
The loss values are shown in the following screenshot:



Then, we will plot the progress of the accuracy values throughout the training:

```
plt.title("Train-Val Accuracy")
plt.plot(range(1,num_epochs+1),metric_hist["train"],label="train")
plt.plot(range(1,num_epochs+1),metric_hist["val"],label="val")
plt.ylabel("Accuracy")
plt.xlabel("Training Epochs")
plt.legend()
plt.show()
```

The accuracy values are shown in the following screenshot:



7. Now, let's train the model with the pre-trained weights:

```
import copy

loss_func = nn.CrossEntropyLoss(reduction="sum")
opt = optim.Adam(resnet18_pretrained.parameters(), lr=1e-4)
lr_scheduler = CosineAnnealingLR(opt, T_max=5, eta_min=1e-6)

params_train={
    "num_epochs": 100,
    "optimizer": opt,
    "loss_func": loss_func,
    "train_dl": train_dl,
    "val_dl": val_dl,
    "sanity_check": False,
    "lr_scheduler": lr_scheduler,
    "path2weights": "./models/resnet18_pretrained.pt",
}

resnet18_pretrained, loss_hist, metric_hist=train_val(resnet18_pretrained, params_train)
```

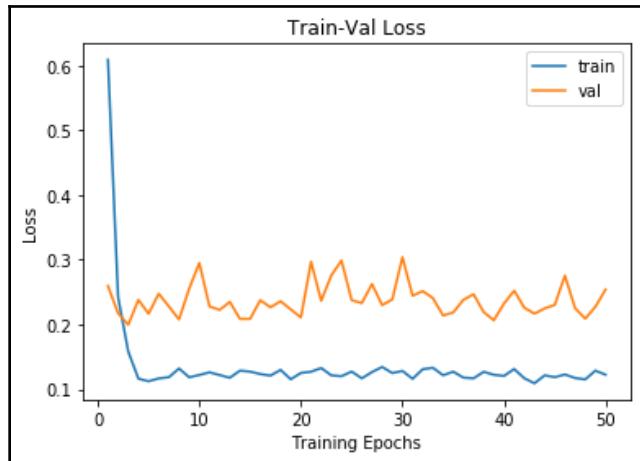
The training will start and you should see its progress:

```
Epoch 0/2, current lr=0.0001
Copied best model weights!
train loss: 0.894398, dev loss: 0.424265, accuracy: 85.38
-----
Epoch 1/2, current lr=0.0001
train loss: 0.437394, dev loss: 0.436810, accuracy: 85.50
-----
Epoch 2/2, current lr=9.05463412215599e-05
Copied best model weights!
train loss: 0.312230, dev loss: 0.379482, accuracy: 87.81
-----
```

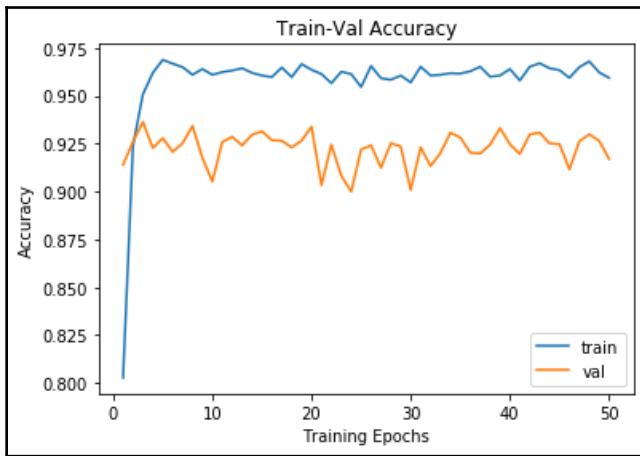
8. Let's plot the training-validation progress:

```
# use the same code as step 6
```

The loss values are shown in the following screenshot:



The accuracy values are shown in the following screenshot:



In the next section, we will explain each step in detail.

How it works...

In *step 1*, we developed a helper function to count the number of correct predictions per batch to be later used for calculating the model accuracy.

In *step 2*, we developed the `loss_batch` helper function. This function receives the loss function and optimizer objects, the model output, and the ground truth label. We only performed the backpropagation step during training.

In *step 3*, the `loss_epoch` helper function receives the model, loss function, dataloader, and optimizer objects. We used the dataloader to fetch batches of data from the dataset. We then moved the batch to the CUDA device and got the model output. We also declared two variables to keep the loss and the number of correct predictions after a batch is executed. You saw how the choice of `reduction="sum"` in the definition of the loss function is deployed to accumulate the loss over the entire dataset. The `sanity_check` flag breaks the loop for the quick execution of the function in the case of debugging.

In *step 4*, we passed the parameters as a Python dictionary. This will improve the code readability. We defined two dictionaries to record the loss and metric values during training for visualization purposes.

An important step is to track the model's performance during training to avoid overfitting. To this end, we evaluated the model on the validation dataset after each training epoch. We then compared the validation loss with the best validation loss: `val_loss < best_loss`. If the validation loss has improved, we will store and make a copy of the weights as the best model weights.

The learning rate scheduler reduces the learning rate using a cosine annealing schedule:

```
# learning rate schedule  
lr_scheduler.step()
```

We recommended a sanity check before starting the actual training. We created a flag called `sanity_check` for this purpose. If set to `True`, the training loop breaks after one batch in an epoch. Therefore, we get to see the whole loop in a short amount of time.

In *step 5*, we trained the model. We saw that the learning rate schedule will reduce the learning rate in each epoch toward `eta_min=1e-6`. As seen, after 10 epochs, the learning is reset with the initial value.

We stored the best model weights (`state_dict`) during training by tracking the model's performance on the validation dataset. The weights are located in the `"./models/weights.pt"` local path. This is useful if the program is stopped for any reason—at least you have a copy of the best weights.

In *step 6*, we saw the plot of loss and accuracy values for both the training and validation datasets. You also saw that training the resnet18 model from scratch does not lead to an impressive performance (validation accuracy is around 0.65). That is because the training data is not sufficient to train such a large model. That is why using pre-trained models or transfer learning can be beneficial.

In *step 7*, we used the pre-trained weights for transfer learning. This will help the model's performance to significantly jump to above 90% even for initial epochs. This is the power of transfer learning. The plots of loss and accuracy values are depicted in *step 8*.

See also

Try to experiment with other built-in models and see how they perform on the STL-10 dataset.

Deploying the model

Let's assume that you want to deploy the model for inference in a new script separate from the training scripts. In this case, the model and weights do not exist in memory. So, we need to construct an object of the model class and then load the stored weights into the model.

How to do it...

We will load the model parameters, verify the model's performance, and deploy the model on the test dataset:

1. Load the resnet18 model:

```
from torch import nn
from torchvision import models

model_resnet18 = models.resnet18(pretrained=False)
num_ftrs = model_resnet18.fc.in_features
num_classes=10
model_resnet18.fc = nn.Linear(num_ftrs, num_classes)
```

2. Let's load state_dict from the stored file into the model:

```
import torch

path2weights="../models/resnet18_pretrained.pt"
model_resnet18.load_state_dict(torch.load(path2weights))
```

3. Then, we will set the model in evaluation mode:

```
model_resnet18.eval()
```

4. Next, we will move the model onto the CUDA device if available:

```
if torch.cuda.is_available():
    device = torch.device("cuda")
    model_resnet18=model_resnet18.to(device)
```

5. Next, we will develop a helper function to deploy the model on a dataset.

The helper function starts with initialization:

```
def deploy_model(model,dataset,device,
num_classes=10,sanity_check=False):

    len_data=len(dataset)
    y_out=torch.zeros(len_data,num_classes)
    y_gt=np.zeros((len_data),dtype="uint8")
    model=model.to(device)
    elapsed_time=[]
```

The helper function continues with a loop over the dataset:

```
with torch.no_grad():
    for i in range(len_data):
        x,y=dataset[i]
        y_gt[i]=y
        start=time.time()
        yy=model(x.unsqueeze(0).to(device))
        y_out[i]=torch.softmax(yy,dim=1)
        elapsed=time.time()-start
        elapsed_times.append(elapsed)

    if sanity_check is True:
        break
```

The helper function ends by returning the outputs and printing the inference time:

```
inference_time=np.mean(elapsed_times)*1000
print("average inference time per image on %s: %.2f ms "
%(device,inference_time))
return y_out.numpy(),y_gt
```

6. Then, we will call the function to deploy the model on the validation dataset:

```
import time
import numpy as np

y_out,y_gt=deploy_model(cnn_model,val_ds,device=device,sanity_check
=False)
print(y_out.shape,y_gt.shape)
```

This snippet will print the following output:

```
average inference time per image on cuda: 3.62 ms
(1600, 10) (1600,)
```

7. Let's calculate the accuracy of the model on the validation dataset using the predicted outputs:

```
from sklearn.metrics import accuracy_score

y_pred = np.argmax(y_out, axis=1)
print(y_pred.shape,y_gt.shape)

acc=accuracy_score(y_pred,y_gt)
print("accuracy: %.2f" %acc)
```

This snippet will print the following output:

```
(1600,) (1600,)
accuracy: 0.94
```

8. Let's deploy the model on test_ds:

```
y_out,y_gt=deploy_model(model_resnet18,test_ds,device=device)

y_pred = np.argmax(y_out, axis=1)
acc=accuracy_score(y_pred,y_gt)
print(acc)
```

This snippet will print the following output:

```
average inference time per image on cuda: 3.47 ms
0.94
```

9. Next, we will display a few sample images from `test_ds`.

We will import the required packages:

```
from torchvision import utils
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
np.random.seed(1)
```

Then, we will define the helper function:

```
def imshow(inp, title=None):
    mean=[0.4467106, 0.43980986, 0.40664646]
    std=[0.22414584, 0.22148906, 0.22389975]
    inp = inp.numpy().transpose((1, 2, 0))
    mean = np.array(mean)
    std = np.array(std)
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
    plt.pause(0.001) # pause a bit so that plots are updated
```

Then, we will create a grid and display it:

```
grid_size=4
rnd_inds=np.random.randint(0,len(test_ds),grid_size)
print("image indices:",rnd_inds)

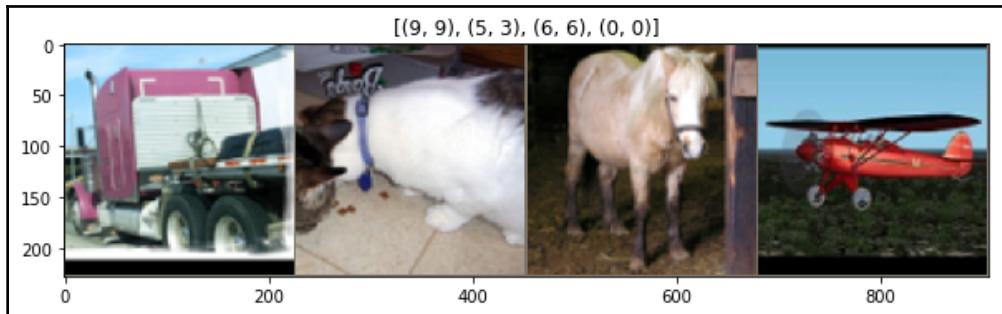
x_grid_test=[test_ds[i][0] for i in rnd_inds]
y_grid_test=[(y_pred[i],y_gt[i]) for i in rnd_inds]

x_grid_test=utils.make_grid(x_grid_test, nrow=4, padding=2)
print(x_grid_test.shape)

plt.rcParams['figure.figsize'] = (10, 5)
imshow(x_grid_test,y_grid_test)

image indices: [5157 235 3980 5192]
torch.Size([3, 100, 394])
```

The sample images and the predictions are shown in the following screenshot:



10. Let's also measure the inference time on the `cpu` device:

```
device_cpu = torch.device("cpu")
y_out,y_gt=deploy_model(model_resnet18,val_ds,device=device_cpu,sanity_check=False)
print(y_out.shape,y_gt.shape)

average inference time per image on cpu: 14.41 ms
(1600, 10) (1600,)
```

In the next section, we will explain each step in detail.

How it works...

In *step 1*, we loaded the built-in model from `torchvision.models`. In *step 2*, we loaded `state_dict`, which contains the model weights in the model. For deployment, it is necessary to set the model to evaluation mode. This is important since some layers (such as dropout and BatchNorm) perform differently in training and deployment modes. If the CUDA device is available, we should move the model onto the CUDA device. The model is ready to be deployed.

In *step 5*, the helper function returns the model outputs and ground truth labels as NumPy arrays for the dataset. Also, the inference time per image on the CUDA device is estimated to be 3.53 ms. Later, we will see the inference time on the CPU device.

In *step 6*, we verified the stored model by deploying it on the validation dataset. Lots of things can go wrong during the development of deep learning models. We recommended verifying the stored model's performance by deploying the model on a known dataset, for instance, the validation dataset.

In *step 7*, we used the `scikit-learn` package to calculate the accuracy of our binary classification model. You can install the package using the following code:

```
# install scikit-learn  
$ conda install scikit-learn
```

This shows an accuracy of `0.94` on the validation dataset. This is a good verification of our model.

In *step 8*, we deployed the model on `test_ds`. We did not use `test_ds` for validation, so it can be considered a hidden dataset to the model. Luckily, we have the labels for `test_ds` too and we can evaluate the model's performance on `test_ds`. This shows an accuracy of `0.95`, which is close to the accuracy of `val_ds`.

In *step 9*, we displayed a few sample images and the predicted labels from `test_ds`.

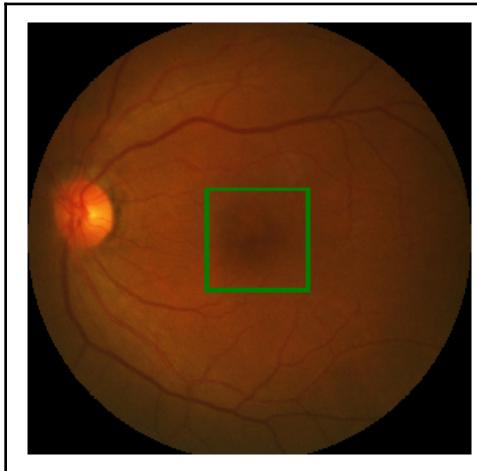
In *step 10*, we wanted to see the deployment time on the CPU. This would be informative if we wanted to deploy the model on devices without a GPU. As seen, the deployment time is around 25 ms per image. This is still a low number, which could be used for many real-time applications.

4

Single-Object Detection

Object detection is the process of finding locations of specific objects in images. Depending on the number of objects in images, we may deal with single-object or multi-object detection problems. This chapter will focus on developing a deep learning model using PyTorch to perform single-object detection. In single-object detection, we are attempting to locate only one object in a given image. The location of the object can be defined by a bounding box.

As an example, the following screenshot depicts the location of the fovea (a small pit) in an eye image using a green bounding box:



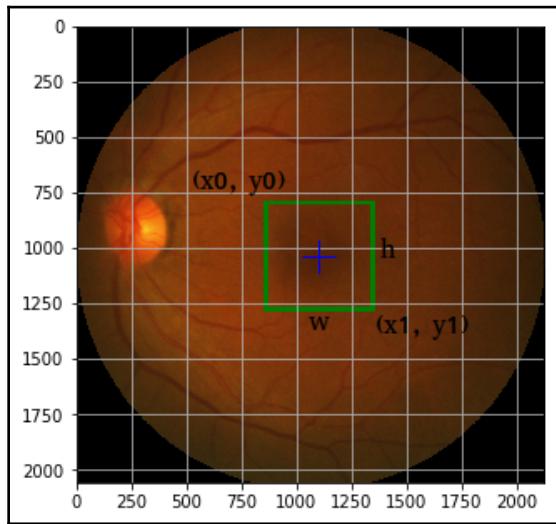
We can represent a bounding box with four numbers in one of the following formats:

- $[x0, y0, w, h]$
- $[x0, y0, x1, y1]$
- $[xc, yc, w, h]$

Here, the preceding elements represent the following:

- $x0, y0$: The coordinates of the top left of the bounding box
- $x1, y1$: The coordinates of the bottom right of the bounding box
- w, h : The width and height of the bounding box
- xc, yc : The coordinates of the centroid of the bounding box

As an example, let's look at the following screenshot in which $xc, yc = 1099, 1035$; $w, h = 500, 500$; $x0, y0 = 849, 785$; and $x1, y1 = 1349, 1285$:



Therefore, the goal of single-object detection will be to predict a bounding box using four numbers. In the case of square objects, we can fix the width and height and simplify the problem to predict just two numbers. In this chapter, we will learn to create an algorithm to locate the fovea in eye images using two numbers.

In this chapter, we will cover the following recipes:

- Exploratory data analysis
- Data transformation for object detection
- Creating custom datasets
- Creating the model
- Defining the loss, optimizer, and IOU metric
- Training and evaluation of the model
- Deploying the model

Exploratory data analysis

Exploratory data analysis is usually performed to understand the characteristics of data. In exploratory data analysis, we will inspect our dataset and visualize samples or statistical features of our data using boxplots, histograms, and other visualization tools. For instance, for tabular data, we would like to see the columns, a few rows, a number of records, and statistical metrics, such as the mean and standard deviation of our data. For imaging data, we would display sample images, labels, or bounding boxes of the objects in the images.

We will use the data from the **iChallenge-AMD** competition on the **Grand Challenge** website. This competition has multiple tasks, including classification, localization, and segmentation. We are only interested in the localization task. In this recipe, we will explore the iChallenge-AMD dataset.

Getting ready

Let's download the iChallenge-AMD dataset:

1. To find the dataset, visit <https://amd.grand-challenge.org/>.

To be able to download the data, you need to create a free account.



2. Select the **Download** section from the sidebar.
3. Click on **images and AMD labels**, as highlighted in the following screenshot, to download the data:

The screenshot shows a 'Download' section with the following options:

- Disc and fovea annotations** (released on Nov 21)
- Training**:
 - Images and AMD labels** (released on Oct 20) - [Link for Mainland China](#) PWD: km0n
 - [Link for Mainland China](#) PWD: mb36
- Lesions annotations** (released on Jan 1)
 - [Link for Mainland China](#) (released on Jan 1)

4. Click on the **Download** button and then select **Direct Download**. The `iChallenge-AMD-Training400.zip` file will be downloaded. Move the `.zip` file to a folder named `data` in the same location as your code.
5. Extract the `.zip` file into a folder named `data/Training400`. The folder should contain another folder named `AMD` with 89 images, a folder named `Non-AMD` with 311 images, and an Excel file named `Fovea_location.xlsx`.
6. The Excel file contains the centroid locations of the fovea in the 400 images (89+311).

Moreover, make sure that `seaborn` is installed in your `conda` environment. You can install `seaborn` using the following command:

```
$ conda install -c anaconda seaborn
```

Now, let's move on to the steps.

How to do it...

To explore the dataset, we begin by loading the labels and displaying sample images:

1. Let's begin by loading `Fovea_location.xlsx` and printing out its head:

```
import os
import pandas as pd

path2data="./data/"

path2labels=os.path.join(path2data,"Training400","Fovea_location.xlsx")

# make sure to install xlrd
labels_df=pd.read_excel(path2labels,index_col="ID")

labels_df.head()
```

The preceding code snippet will print out the following table:

	imgName	Fovea_X	Fovea_Y
ID			
1	A0001.jpg	1182.264278	1022.018842
2	A0002.jpg	967.754046	1016.946655
3	A0003.jpg	1220.206714	989.944033
4	A0004.jpg	1141.140888	1000.594955
5	A0005.jpg	1127.371832	1071.109440

2. Next, we print out the tail of the Excel file:

```
labels_df.tail()
```

This will print out the following table:

	imgName	Fovea_X	Fovea_Y
ID			
396	N0307.jpg	823.024991	690.210211
397	N0308.jpg	647.598978	795.653188
398	N0309.jpg	624.571803	755.694880
399	N0310.jpg	687.523044	830.449187
400	N0311.jpg	746.107631	759.623062

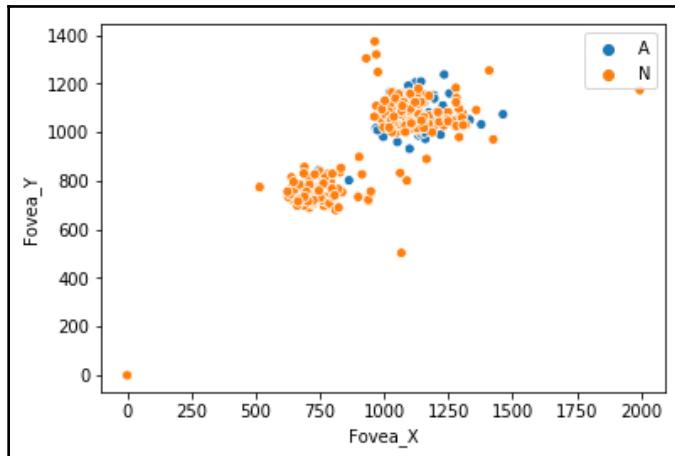
3. Then, we will show the scatter plot of the Fovea_X and Fovea_Y coordinates:

Import seaborn and plot the coordinates:

```
import seaborn as sns
%matplotlib inline

AorN=[imn[0] for imn in labels_df.imgName]
sns.scatterplot(x=labels_df['Fovea_X'],
y=labels_df['Fovea_Y'],hue=AorN)
```

This will display the scatter plot as shown in the following screenshot:



4. Next, we will show a few sample images. We will import the required packages and fix the random seed:

```
import numpy as np
from PIL import Image, ImageDraw
import matplotlib.pyplot as plt

# fix random seed
np.random.seed(2019)
```

Then, we will set the plot parameters:

```
plt.rcParams['figure.figsize'] = (15, 9)
plt.subplots_adjust(wspace=0, hspace=0.3)
nrows, ncols = 2, 3
```

Then, we will select a random set of image ids:

```
imgName=labels_df["imgName"]
ids=labels_df.index
rndIds=np.random.choice(ids, nrows*ncols)
print(rndIds)
```

The preceding code snippet will print the following list of image ids:

```
[ 73 371 160 294 217 191 247 25]
```

Next, we will define a helper function to load an image and its label from the local files:

```
def load_img_label(labels_df,id_):
    imgName=labels_df["imgName"]
    if imgName[id_][0]=="A":
        prefix="AMD"
    else:
        prefix="Non-AMD"
    fullPath2img=os.path.join(path2data,"Training400",prefix,imgName[id_])
    img = Image.open(fullPath2img)
    x=labels_df["Fovea_X"][id_]
    y=labels_df["Fovea_Y"][id_]
    label=(x,y)
    return img,label
```

Next, we will define a helper function to show the image and label as a bounding box:

```
def show_img_label(img,label,w_h=(50,50),thickness=2):
    w,h=w_h
    cx,cy=label
    draw = ImageDraw.Draw(img)
    draw.rectangle(((cx-w/2, cy-h/2), (cx+w/2, cy+h/2)), outline="green", width=thickness)
    plt.imshow(np.asarray(img))
```

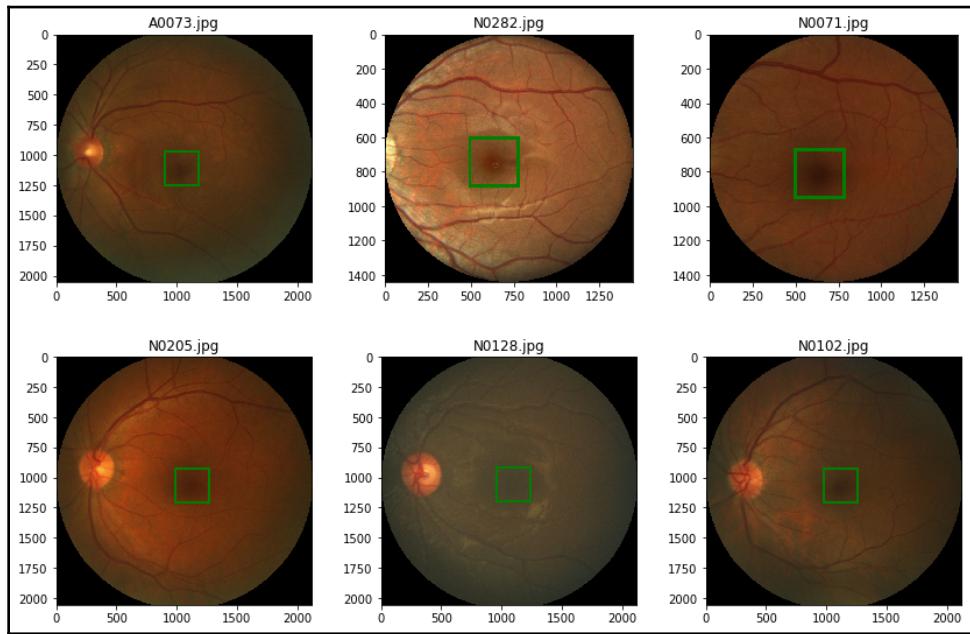
Then, we will show the selected images together with the fovea bounding boxes:

```
for i,id_ in enumerate(rndIds):
    img,label=load_img_label(labels_df,id_)
    print(img.size,label)
    plt.subplot(nrows, ncols, i+1)
    show_img_label(img,label,w_h=(150,150),thickness=20)
    plt.title(imgName[id_])
```

The preceding code snippet will print the following output:

```
(2124, 2056) (1037.89889229694, 1115.71768088143)
(1444, 1444) (635.148992978281, 744.648850248249)
(1444, 1444) (639.360312038611, 814.762764100936)
(2124, 2056) (1122.08407442503, 1067.58829793991)
(2124, 2056) (1092.93333646222, 1055.15333296773)
(2124, 2056) (1112.50135915347, 1070.7251775623)
```

Also, the following screenshot shows the sample images with the fovea bounding boxes:



5. Next, we will collect the image widths and heights in two lists:

```

h_list, w_list = [], []
for id_ in ids:
    if imgName[id_][0] == "A":
        prefix = "AMD"
    else:
        prefix = "Non-AMD"
    fullPath2img = os.path.join(path2data, "Training400", prefix, imgName[id_])
    img = Image.open(fullPath2img)
    h, w = img.size
    h_list.append(h)
    w_list.append(w)

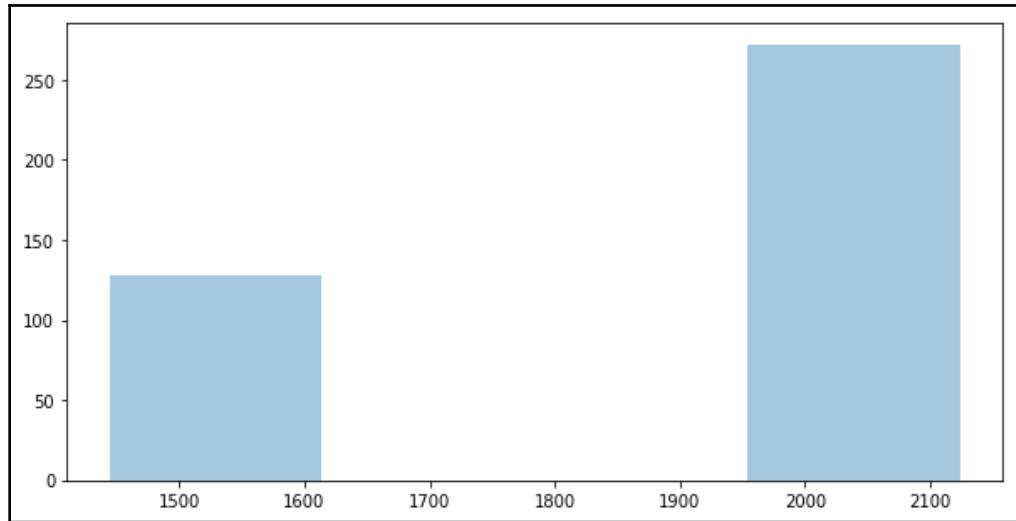
```

6. Then, we will plot the distributions of heights and widths:

Plot the distribution of the image heights as follows:

```
sns.distplot(a=h_list, kde=False)
```

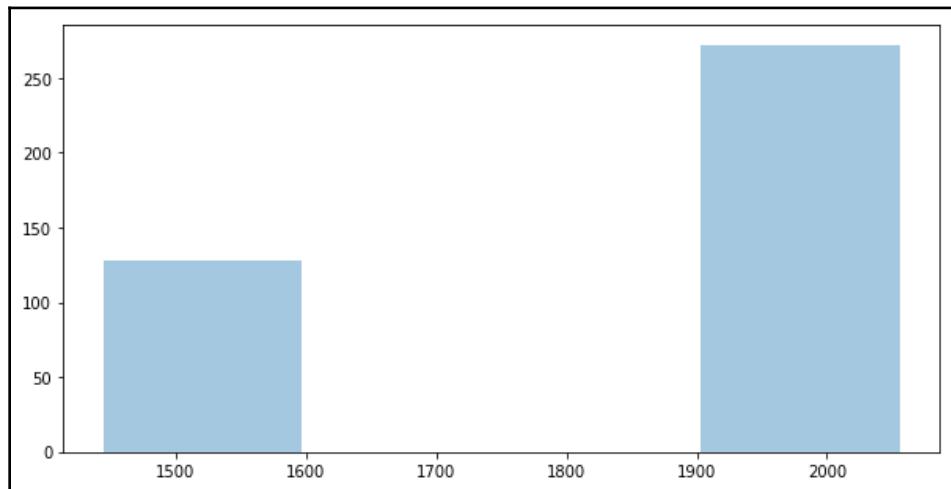
The preceding code will depict the following screenshot:



Next, we will plot the distribution of the image widths:

```
sns.distplot(a=w_list, kde=False)
```

The preceding code will depict the following screenshot:



In the next section, we will explain each step in detail.

How it works...

In *step 1*, we loaded the labels from the Excel file. We used the pandas library and assumed that `Fovea_location.xlsx` is located in the `./data/Training400` folder. On Windows machines, you might not need to install anything. For Linux machines, to load Excel files using pandas, install the `xlrd` package in your `conda` environment using the following command:

```
$ conda install -c anaconda xlrd
```

Then, we loaded the file into a pandas DataFrame and printed out its head. The DataFrame has three columns: `imgName`, `Fovea_X`, and `Fovea_Y`. The `imgName` column refers to the image filename, while `Fovea_X` and `Fovea_Y` represent the x and y coordinates of the fovea center. Note that the first 89 rows correspond to the AMD images and start with the letter A.

In *step 2*, we printed out the tail of the DataFrame. You can see that `ID` reaches 400, the same as the number of images. Also, note that the last 311 rows correspond to the Non-AMD images and start with the letter N.

In *step 3*, we showed the scatterplot of `Fovea_X` and `Fovea_Y`. We used the seaborn library to show the scatterplot. Seaborn is a Python data visualization library based on `matplotlib`. It provides a high-level interface for drawing attractive and informative statistical graphics.

As has been observed, the centers are clustered into two groups. Also, there is no correlation between the image class (AMD versus Non-AMD) and the fovea location. It is also interesting to see that there are a few images with the fovea at the $(0, 0)$ coordinates. We can drop zero values from the DataFrame, as shown in the following code block:

```
labels_df=labels_df.replace(0,None)
labels_df.dropna
```

However, since there are only a few images like this, we opted to keep them.

In *step 4*, we displayed a few random images with the fovea bounding boxes. The fovea location is given as a center point. Therefore, to show a bounding box, we create a rectangle using `rectangle` from the `PIL.imageDraw` package at the center of the fovea. We also printed the image sizes. As we saw, images have different sizes.

In *step 5*, we get lists of image heights and widths. This will be used to plot the distributions of heights and widths in *step 6*. The plots of distributions reveal that the majority of heights and width are in the range of 1900 to 2100.

Data transformation for object detection

Data augmentation and transformation is a critical step in training deep learning algorithms, especially for small datasets. The iChallenge-AMD dataset in this chapter has only 400 images, which is considered a small dataset. As a reminder, we will later split 20 percent of this dataset for evaluation purposes. Since the images have different sizes, we need to resize all images to a pre-determined size. Then, we can utilize a variety of augmentation techniques, such as horizontal flipping, vertical flipping, and translation, to expand our dataset during training.

In object detection tasks, when we perform such transformations on images, we also need to update the labels. For instance, when we flip an image horizontally, the location of objects in the image will change. While `torchvision.transforms` provides utility functions for image transformations, we need to build our own functions for updating the labels. In this recipe, we will develop a pipeline for transforming images and labels in single-object detection. We will develop horizontal flipping, vertical flipping, translation, and resizing. You can then add more transformations to the pipeline as you need.

How to do it...

We will create a data transformation pipeline for single-object detection:

1. First, we will define a helper function to resize images:

```
import torchvision.transforms.functional as TF

def resize_img_label(image,label=(0.,0.),target_size=(256,256)):
    w_orig,h_orig = image.size
    w_target,h_target = target_size
    cx, cy= label
    image_new = TF.resize(image,target_size)
    label_new= cx/w_orig*w_target, cy/h_orig*h_target
    return image_new,label_new
```

Let's try resizing an image using the preceding function:

```
img, label=load_img_label(labels_df,1)
print(img.size,label)

img_r,label_r=resize_img_label(img,label)
print(img_r.size,label_r)

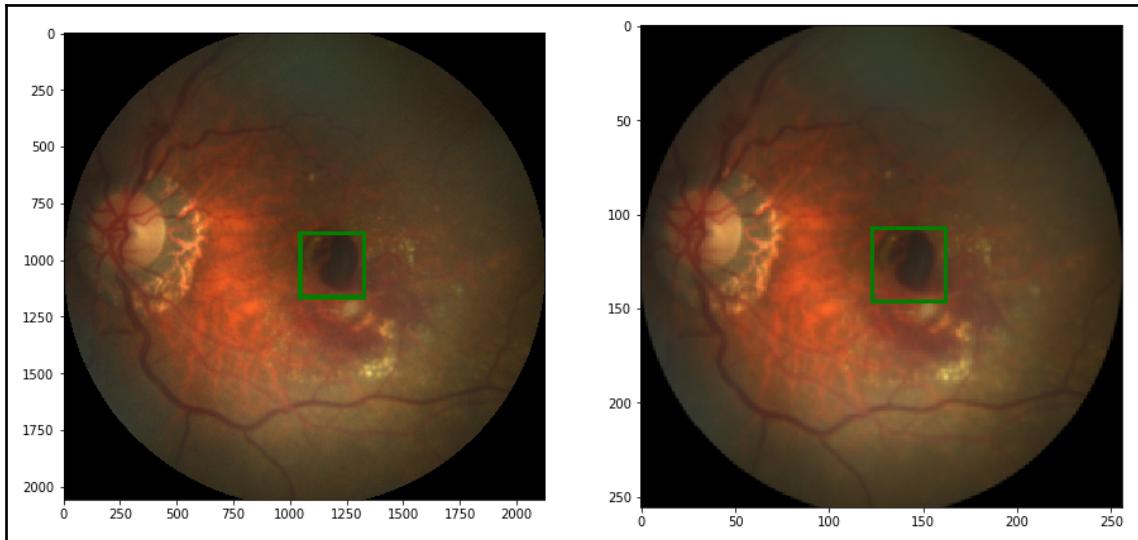
plt.subplot(1,2,1)
show_img_label(img,label,w_h=(150,150),thickness=20)
```

```
plt.subplot(1,2,2)
show_img_label(img_r,label_r)
```

The preceding code snippet will print the original and resized image sizes and labels:

```
(2124, 2056) (1182.26427759023, 1022.01884158854)
(256, 256) (142.4951295024006, 127.25526432230848)
```

Also, the preceding snippet will depict the original and resized images and the fovea bounding boxes, as in the following screenshot:



2. Next, we will define a helper function to randomly flip images horizontally:

```
def random_hflip(image,label):
    w,h=image.size
    x,y=label

    image = TF.hflip(image)
    label = w-x, y
    return image,label
```

Let's try flipping an image using the preceding function:

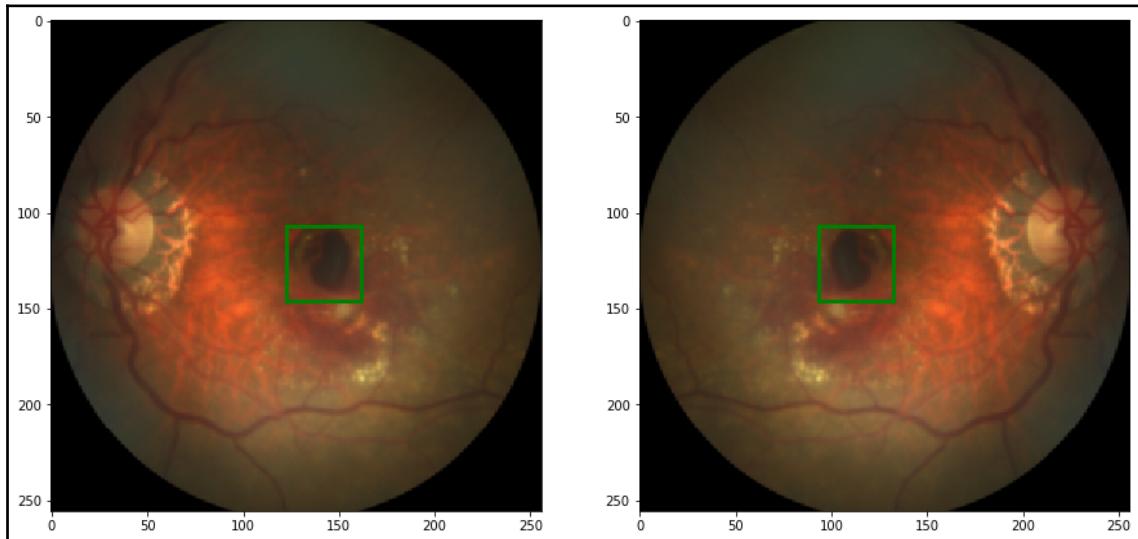
```
img, label=load_img_label(labels_df,1)

img_r,label_r=resize_img_label(img,label)
```

```
img_fh,label_fh=random_hflip(img_r,label_r)

plt.subplot(1,2,1)
show_img_label(img_r,label_r)
plt.subplot(1,2,2)
show_img_label(img_fh,label_fh)
```

The preceding code snippet will display the image before and after horizontal flipping, as in the following screenshot:



3. Next, we will define a function to randomly flip images vertically:

```
def random_vflip(image,label):
    w,h=image.size
    x,y=label

    image = TF.vflip(image)
    label = x, w-y
    return image, label
```

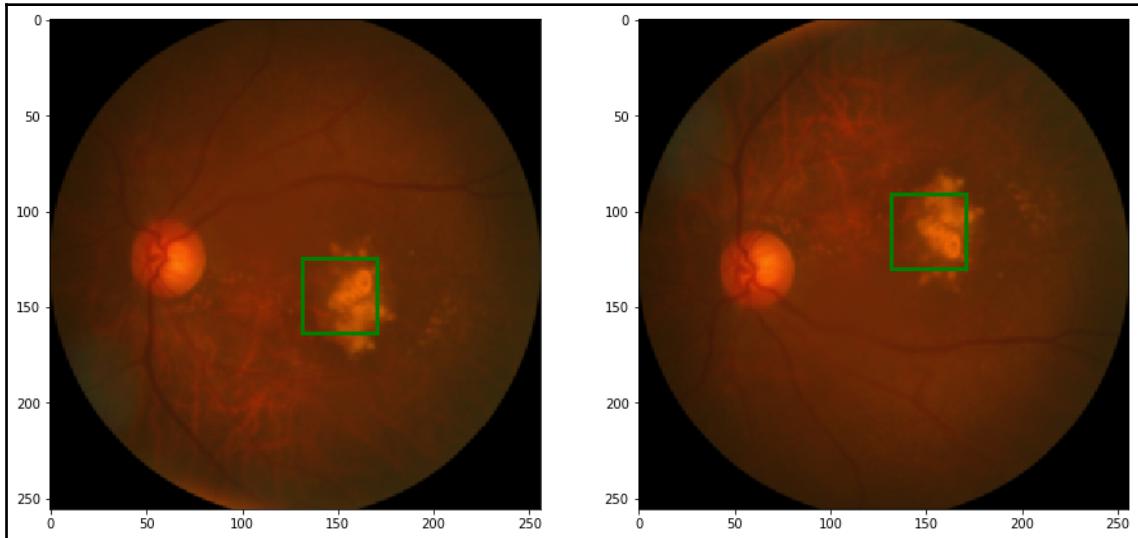
Let's try flipping an image using the preceding function:

```
img, label=load_img_label(labels_df,7)
img_r,label_r=resize_img_label(img,label)
img_fv,label_fv=random_vflip(img_r,label_r)

plt.subplot(1,2,1)
show_img_label(img_r,label_r)
```

```
plt.subplot(1,2,2)
show_img_label(img_fv,label_fv)
```

This will display the image before and after vertical flipping, as in the following screenshot:



4. Next, we will define a helper function to randomly shift or translate images in either direction:

```
import numpy as np
np.random.seed(1)

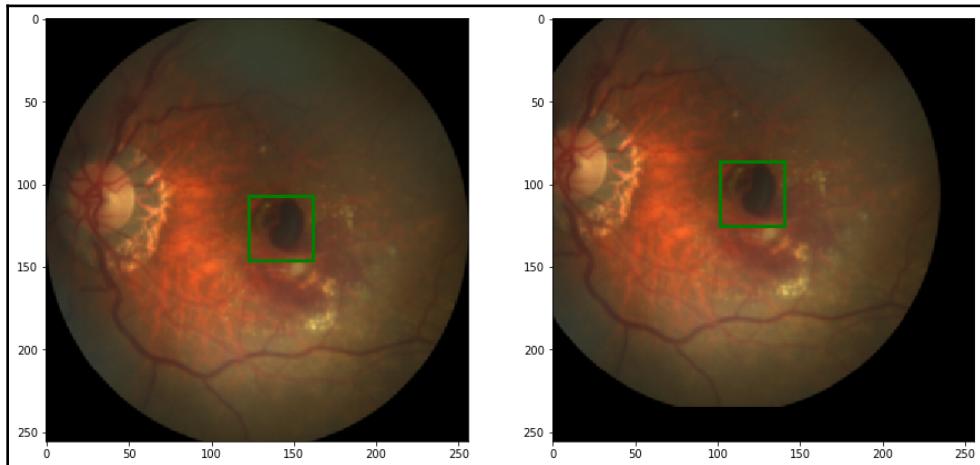
def random_shift(image,label,max_translate=(0.2,0.2)):
    w,h=image.size
    max_t_w, max_t_h=max_translate
    cx, cy=label
    trans_coef=np.random.rand()*2-1
    w_t = int(trans_coef*max_t_w*w)
    h_t = int(trans_coef*max_t_h*h)
    image=TF.affine(image,translate=(w_t,
    h_t),shear=0,angle=0,scale=1)
    label = cx+w_t, cy+h_t
    return image,label
```

Let's try translating an image using the `random_shift` function:

```
img, label=load_img_label(labels_df,1)
img_r,label_r=resize_img_label(img,label)
img_t,label_t=random_shift(img_r,label_r,max_translate=(.5,.5))

plt.subplot(1,2,1)
show_img_label(img_r,label_r)
plt.subplot(1,2,2)
show_img_label(img_t,label_t)
```

The preceding code snippet will display the image before and after translation, as in the following screenshot:



5. Next, we will compose multiple transformations into one function to define `transformer`:

```
def transformer(image, label, params):
    image,label=resize_img_label(image,label,params["target_size"])
    if random.random() < params["p_hflip"]:
        image,label=random_hflip(image,label)
    if random.random() < params["p_vflip"]:
        image,label=random_vflip(image,label)
    if random.random() < params["p_shift"]:
        image,label=random_shift(image,label,
        params["max_translate"])
    image=TF.to_tensor(image)
    return image, label
```

Let's try to transform an image using transformer:

```
import random
np.random.seed(0)
random.seed(0)

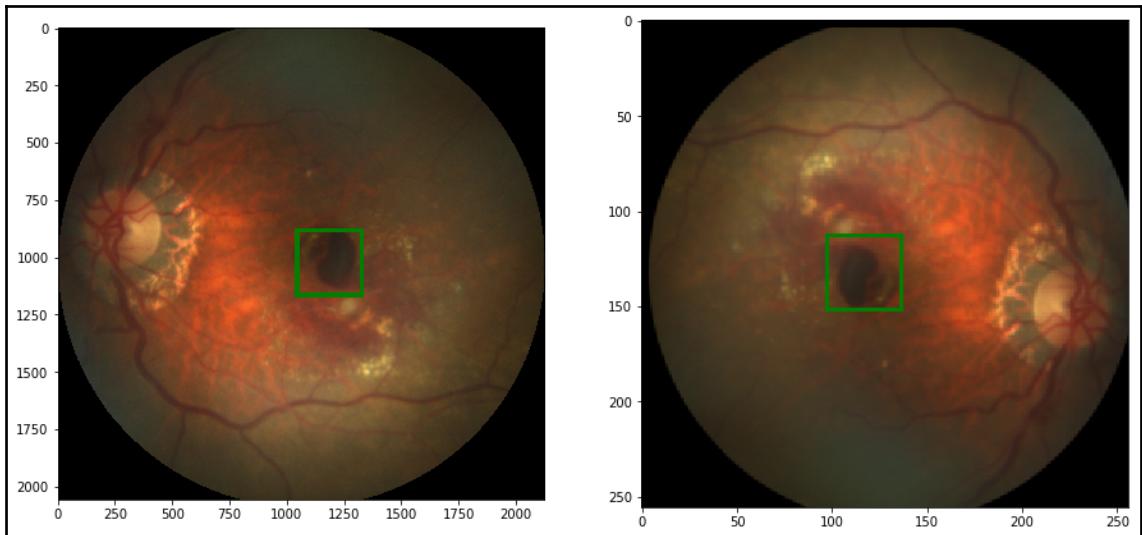
img, label=load_img_label(labels_df,1)

params={
    "target_size" : (256, 256),
    "p_hflip" : 1.0,
    "p_vflip" : 1.0,
    "p_shift" : 1.0,
    "max_translate": (0.2, 0.2),
}
img_t,label_t=transformer(img,label,params)
```

Then, we will show the original and transformed images:

```
plt.subplot(1,2,1)
show_img_label(img,label,w_h=(150,150),thickness=20)
plt.subplot(1,2,2)
show_img_label(TF.to_pil_image(img_t),label_t)
```

This will display the original and transformed images, as in the following screenshot:



In the next section, we will explain each step in detail.

How it works...

In *step 1*, we defined a helper function to resize PIL images and update the labels to a target size. For resizing PIL images, we can use the `resize` function from `torchvision.transforms.functional`. As a reminder, PIL returns the image size in this format: `width, height = image.size`. This could be confusing and is sometimes a source of bugs if you worked with the OpenCV package.



PIL returns the image size in the format of `width, height = image.size`.

To update the label, the fovea coordinates, we need to scale the coordinate by the factor of resizing in each dimension. This factor will be `w_target/w_orig` and `h_target/h_orig` for the `x` and `y` coordinates, respectively. To load and show the sample image and label, we use the helper functions developed in the *Exploratory data analysis* recipe.

In *step 2*, we built a helper function to horizontally flip images and labels. We used the `hflip` function from `torchvision` to horizontally flip images. In the case of a horizontal flip, the `y` coordinate remains the same and only the `x` coordinate of the fovea location will change to `width - x`. Check out the image before and after flipping and notice how the eye disk is moved from the left side to the right side of the image.

In *step 3*, we built a helper function to vertically flip images and labels. We used the `vflip` function from `torchvision` to vertically flip images. In the case of vertical flipping, the `x` coordinate remains the same and only the `y` coordinate of the fovea location will change to `height - y`. Check out the image before and after flipping and notice how the location of the fovea and eye disk are changed.

In *step 4*, we defined a function to shift or translate images to the left, right, up, or down. The amount of translation is chosen randomly; however, it is bounded by the `max_translate = (0.2, 0.2)` parameter. This means that the maximum translation in `x` and `y` dimensions will be $0.2 * \text{width}$ and $0.2 * \text{height}$. For example, for an image size of $256 * 256$, the maximum translation will be 51 pixels in each direction. To set the maximum image translation during data augmentation, consider the location of the object of interest. You do not want the object to fall outside the image after translation. For most problems, a value in the range of $[0.1, 0.2]$ is safe. However, make sure to adjust this value according to your specific problem. Also, to randomly translate the image in either direction (left, right, up, or down), we generate a random value in the range of $[-1, 1]$ and multiply it by the maximum translation.

To translate images, we use the `affine` function from `torchvision`. This function can perform other types of transformations, such as rotation, shearing, and scaling. Here, we only used the translation feature. Lastly, both x and y coordinates of the fovea center could change by the amount of translation.

In *step 5*, we stacked multiple transformations into one function. We will later pass the transformation function to the dataset class. As has been observed from the transformed images, five transformations are sequentially applied to the `PIL` image. You should pay attention to the order of the functions when building custom transformations. For instance, it is better to resize the image first to reduce the computational complexity of other transformations. Moreover, converting to tensors using `TF.to_tensor` goes at the end. This transformation scales `PIL` images to the range of `[0, 1]` and reshapes images into the `[channel, height, width]` shape. As a result, to display the transformed image, we use `TF.to_pil_image()` to convert it back to a `PIL` image. We also parameterized the augmentation parameters so that you can play with different values. If you want to disable a transformation, you can simply set the probability to zero, as we will do for the validation dataset. A common value for the probability of transformation is `0.5`. Here, to force all transformations, we set the probabilities to `1.0`.

There's more...

There are other types of transformations that you can apply to images for data augmentation. Some of them do not require any update to the label.

For example, we can create new images by adjusting the brightness, as shown in the following code block:

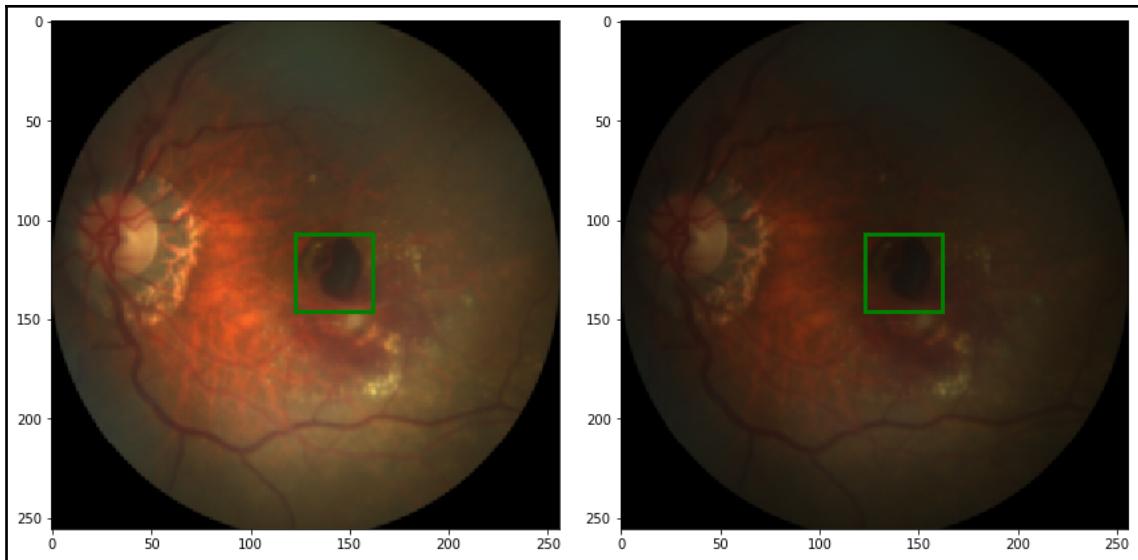
```
img, label=load_img_label(labels_df, 1)

# resize image and label
img_r,label_r=resize_img_label(img,label)

# adjust brightness
img_t=TF.adjust_brightness(img_r,brightness_factor=0.5)
label_t=label_r

plt.subplot(1,2,1)
show_img_label(img_r,label_r)
plt.subplot(1,2,2)
show_img_label(img_t,label_t)
```

The original and transformed images are shown in the following screenshot:



Similarly, we can create new images by adjusting the contrast and gamma correction, as shown in the following code block:

```
# brightness  
img_t=TF.adjust_contrast(img_r,contrast_factor=0.4)  
  
# gamma correction  
img_t=TF.adjust_gamma(img_r,gamma=1.4)
```

There is another transformation that we usually perform on the labels. In this transformation, we scale the labels to the range of [0, 1] using the following function:

```
def scale_label(a,b):  
    div = [ai/bi for ai,bi in zip(a,b)]  
    return div
```

For objection detection tasks, it is important to scale the labels to the range of [0, 1] for better model convergence.



We can integrate these transformations into the `transformer` function, as shown in the following code block:

```
def transformer(image, label, params):  
    # previous transformations here  
    if random.random() < params["p_brightness"]:
```

```
brightness_factor=1+(np.random.rand()*2-1)*params["brightness_factor"]
    image=TF.adjust_brightness(image,brightness_factor)

if random.random() < params["p_contrast"]:
    contrast_factor=1+(np.random.rand()*2-1)*params["contrast_factor"]
    image=TF.adjust_contrast(image,contrast_factor)

if random.random() < params["p_gamma"]:
    gamma=1+(np.random.rand()*2-1)*params["gamma"]
    image=TF.adjust_gamma(image,gamma)

if params["scale_label"]:
    label=scale_label(label,params["target_size"])
image=TF.to_tensor(image)
return image, label
```

Let's try to transform an image using the latest `transformer` function.

Load the image and label:

```
np.random.seed(0)
random.seed(0)

# load image and label
img, label=load_img_label(labels_df,1)
```

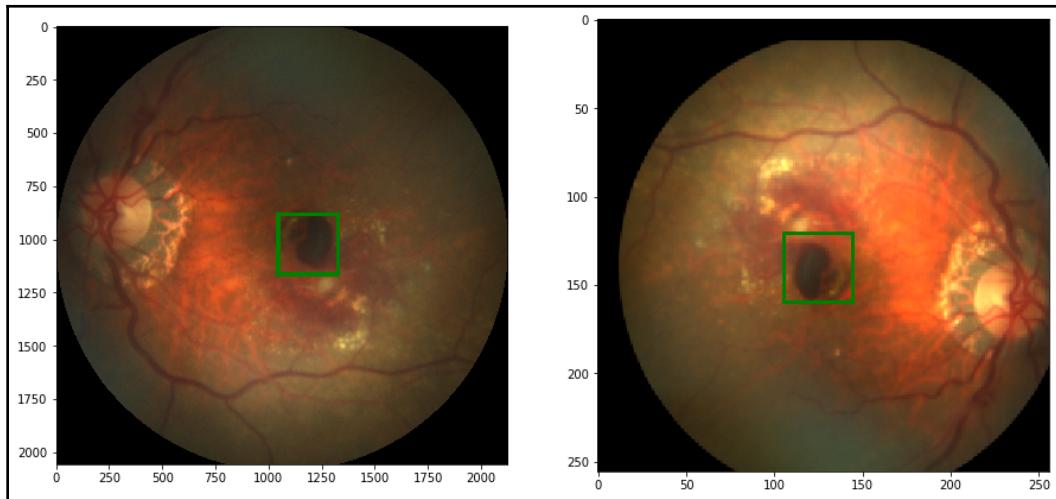
Set the transformation parameters and then apply the transformations:

```
params={
    "target_size" : (256, 256),
    "p_hflip" : 1.0,
    "p_vflip" : 1.0,
    "p_shift" : 1.0,
    "max_translate": (0.5, 0.5),
    "p_brightness": 1.0,
    "brightness_factor": 0.8,
    "p_contrast": 1.0,
    "contrast_factor": 0.8,
    "p_gamma": 1.0,
    "gamma": 0.4,
    "scale_label": False,
}
img_t,label_t=transformer(img,label,params)
```

Then, show the original and transformed images by executing the following code snippet:

```
plt.subplot(1,2,1)
show_img_label(img,label,w_h=(150,150),thickness=20)
plt.subplot(1,2,2)
show_img_label(TF.to_pil_image(img_t),label_t)
```

You can check out the original and transformed images in the following screenshot:



Lastly, to rescale the labels back to the image size, use the following helper function:

```
def rescale_label(a,b):
    div = [ai*bi for ai,bi in zip(a,b)]
    return div
```

In the next section, we will explain each step in detail.

See also

There are other open source packages that will provide a variety of built-in transformations. We have provided a few links in the following list. You can check them out and try to integrate them into your pipeline for data augmentation:

- Augmenter: <https://github.com/mdbloice/Augmentor>
- imgaug: <https://github.com/aleju/imgaug>
- Albumentations: <https://github.com/albu/albumentations>

Creating custom datasets

In this recipe, we will use the `Dataset` class from `torch.utils.data` to create custom datasets for loading and processing data. We can do this by sub-classing the `Dataset` class and overriding `__init__` and the `__getitem__` functions. The `__len__` function returns the dataset length and is callable with the Python `len` function. The `__getitem__` function returns an image at the specified index. Then, we will use the `Dataloader` class from `torch.utils.data` to create data loaders. Using data loaders, we can automatically get mini-batches of data for processing.

How to do it...

We will create the training and validation datasets and the data loaders.

1. Let's define a custom dataset class. First, we will load the required packages:

```
from torch.utils.data import Dataset  
from PIL import Image
```

Then, we will define the bulk of the dataset class:

```
class AMD_dataset(Dataset):  
    def __init__(self, path2data, transform, trans_params):  
        pass  
    def __len__(self):  
        # return size of dataset  
        return len(self.labels)  
    def __getitem__(self, idx):  
        pass
```

Next, we will define the `__init__` function:

```
def __init__(self, path2data, transform, trans_params):  
  
    path2labels=os.path.join(path2data,"Training400_labels","Fovea_loca  
    tion.xlsx")  
    labels_df=pd.read_excel(path2labels,index_col="ID")  
    self.labels = labels_df[["Fovea_X","Fovea_Y"]].values  
  
    self.imgName=labels_df["imgName"]  
    self.ids=labels_df.index  
  
    self.fullPath2img=[0]*len(self.ids)  
    for id_ in self.ids:  
        if self.imgName[id_][0]=="A":
```

```

        prefix="AMD"
    else:
        prefix="Non-AMD"
self fullPath2img[id_-1]=os.path.join(path2data,"Training400",prefi
x,self.imgName[id_])

    self.transform = transform
    self.trans_params=trans_params

```

Next, we will define the `__getitem__` function:

```

def __getitem__(self, idx):
    image = Image.open(self.fullPath2img[idx])
    label= self.labels[idx]
    image,label = self.transform(image,label,self.trans_params)

    return image, label

```

Then, we will override the dataset class functions:

```

AMD_dataset.__init__=__init__
AMD_dataset.__getitem__=__getitem__

```

2. Next, we will create two objects of `AMD_dataset`:

First, we will define the transformation parameters for the training dataset:

```

trans_params_train={
    "target_size" : (256, 256),
    "p_hflip" : 0.5,
    "p_vflip" : 0.5,
    "p_shift" : 0.5,
    "max_translate": (0.2, 0.2),
    "p_brightness": 0.5,
    "brightness_factor": 0.2,
    "p_contrast": 0.5,
    "contrast_factor": 0.2,
    "p_gamma": 0.5,
    "gamma": 0.2,
    "scale_label": True,
}

```

Then, we will define the transformation parameters for the validation dataset:

```

trans_params_val={
    "target_size" : (256, 256),
    "p_hflip" : 0.0,
    "p_vflip" : 0.0,
    "p_shift" : 0.0,
}

```

```
    "p_brightness": 0.0,  
    "p_contrast": 0.0,  
    "p_gamma": 0.0,  
    "gamma": 0.0,  
    "scale_label": True,  
}
```

Then, two objects of the `AMD_dataset` class will be defined:

```
amd_ds1=AMD_dataset(path2data,transformer,trans_params_train)  
amd_ds2=AMD_dataset(path2data,transformer,trans_params_val)
```

3. Next, we will split the dataset into **training** and **validation** sets.

First, we will split the image indices into two groups:

```
from sklearn.model_selection import ShuffleSplit  
  
sss = ShuffleSplit(n_splits=1, test_size=0.2, random_state=0)  
  
indices=range(len(amd_ds1))  
  
for train_index, val_index in sss.split(indices):  
    print(len(train_index))  
    print("-"*10)  
    print(len(val_index))
```

The results are printed in the following code block:

```
320  
-----  
80
```

Then, we will define the training and validation datasets:

```
from torch.utils.data import Subset  
  
train_ds=Subset(amd_ds1,train_index)  
print(len(train_ds))  
  
val_ds=Subset(amd_ds2,val_index)  
print(len(val_ds))
```

The length of the datasets will be printed as in the following code block:

```
320  
80
```

4. Let's now show a sample image from `train_ds` and `val_ds`.

First, we will import the required packages:

```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

# fix random seed
np.random.seed(0)
```

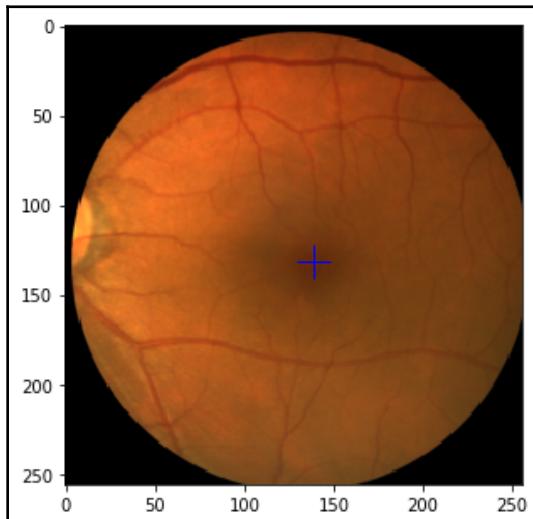
Then, we will define a function to show a tensor image and its label:

```
def show(img,label=None):
    npimg = img.numpy().transpose((1,2,0))
    plt.imshow(npimg)
    if label is not None:
        label=rescale_label(label,img.shape[1:])
        x,y=label
        plt.plot(x,y,'b+',markersize=20)
```

Then, we will show a sample image from `train_ds`:

```
plt.figure(figsize=(5,5))
for img,label in train_ds:
    show(img,label)
    break
```

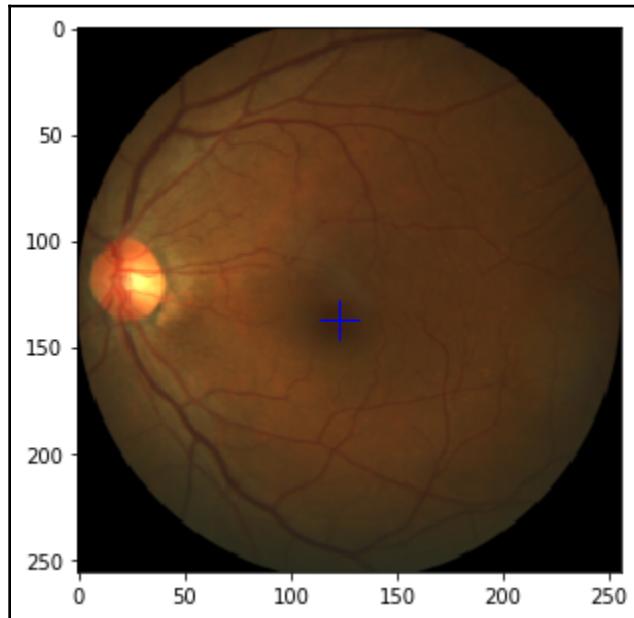
The image and its label are shown in the following screenshot:



Similarly, we can show a sample image from `val_ds`:

```
plt.figure(figsize=(5,5))
for img,label in val_ds:
    show(img,label)
    break
```

The image and its label are shown in the following screenshot:



5. Next, we will define two data loaders for the training and validation datasets:

```
from torch.utils.data import DataLoader
train_dl = DataLoader(train_ds, batch_size=8, shuffle=True)
val_dl = DataLoader(val_ds, batch_size=16, shuffle=False)
```

6. Let's get a batch of data from `train_dl`:

To get a batch, we can break the following loop after one iteration:

```
for img_b, label_b in train_dl:
    print(img_b.shape, img_b.dtype)
    print(label_b)
    break
```

This will print out the following results:

```
torch.Size([8, 3, 256, 256]) torch.float32
[tensor([0.5291, 0.4909, 0.4503, 0.6669, 0.6911, 0.5623, 0.5050,
0.6388],
      dtype=torch.float64), tensor([0.4875, 0.5098, 0.3617,
0.7018, 0.7039, 0.4745, 0.4944, 0.6458],
      dtype=torch.float64)]
```

Notice that the label batch is returned as a **list**. Thus, we will need to convert the list to a tensor, as shown in the following code block:

```
import torch

# extract a batch from training data
for img_b, label_b in train_dl:
    print(img_b.shape, img_b.dtype)

    # convert list to tensor
    label_b=torch.stack(label_b,1)
    label_b=label_b.type(torch.float32)
    print(label_b.shape, label_b.dtype)
    break
```

The preceding snippet will print out the following output:

```
torch.Size([8, 3, 256, 256]) torch.float32
torch.Size([8, 2]) torch.float32
```

7. Similarly, we will get a batch from `val_dl`:

```
for img_b, label_b in val_dl:
    print(img_b.shape, img_b.dtype)

    # convert to tensor
    label_b=torch.stack(label_b,1)
    label_b=label_b.type(torch.float32)
    print(label_b.shape, label_b.dtype)
    break
```

The preceding code snippet will print the following output:

```
torch.Size([16, 3, 256, 256]) torch.float32
torch.Size([16, 2]) torch.float32
```

In the next section, we will explain each step in detail.

How it works...

In *step 1*, we defined the custom dataset class. For better code readability, we presented this step in a few snippets. We first defined the bulk of the dataset class. Then, we implemented the `__init__` function. In this function, we loaded the fovea coordinates from `Fovea_location.xlsx` into `self.labels` to be used as labels. To be able to load the images from local files, we also get the full path to images in `self.fullPath2img`. As we found out in the *Exploratory data analysis* recipe, 89 images are in the `AMD` folder and their names start with the letter `A`, and the remainder are in the `Non-AMD` folder and their names start with the letter `N`. Therefore, we used the first letter to find the image folder and set `prefix` accordingly. Lastly, we set the transformer and transformation parameters. Next, we defined the `__getitem__` function. In this function, we load an image and its label and then transform them using the `transformer` function. In the end, we overrode the two class functions with `__init__` and `__getitem__`.

In *step 2*, we created two objects of the `AMD_dataset` class. This is for data-splitting purposes and will be cleared in the next step. We defined two transformation parameters, `trans_params_train` and `trans_params_val`, and passed them to the two instances, respectively. `trans_params_train` defined the transformations that we want to apply to the training dataset. Thus, we enabled all of the data transformation functions implemented in the *Data transformation for object detection* recipe with the probability of `0.5`. On the other hand, `trans_params_val` defined the transformations that we want to apply to the validation dataset. Thus, we disabled all data augmentation functions by setting their probabilities to `0.0`. Only the resizing of images to `256*256` and the scaling of labels to the range of `[0, 1]` were enabled for both the training and validation datasets.

In *step 3*, we created the training and validation datasets by splitting images into two groups. To this end, we first split the image indices using `ShuffleSplit` from `sklearn`. We assigned 20 percent of the images to the validation dataset. This resulted in 320 images for training and 80 for evaluation. Then, we utilized the `Subset` class from `torch.utils` to subset `train_ds` from the `amd_ds1` dataset at the `train_index` indices. Similarly, we subset `val_ds` from the `amd_ds2` dataset at the `val_index` indices. Note that `train_ds` inherits the `transformer` function of `amd_ds1`, while `val_ds` inherits the `transformer` function of `amd_ds2`. By now, it should be clear to you why we defined two instances of the `AMD_dataset` class in the first place: to have different transformations for the training and validation datasets. If you only define one object of the dataset class and pass different indices, they will have the same transformation function.



If you want to split one PyTorch dataset into two training and validation datasets using `Subset` or `random_split` from `torch.utils.data`, define two objects of the dataset class. If you only define one instance of the dataset class and create subset datasets by passing the difference indices, they will get the same transformation function.

In *step 4*, we depicted a sample image from both `train_ds` and `val_ds`. We defined a helper function to display a tensor image together with its label as a + marker. The tensor was in the $C \times H \times W$ shape so we reshaped it to $H \times W \times C$. The label contains the rescaled x and y coordinates. Thus, we rescaled it back to the image size. We first depicted a sample image from `train_ds`. If you rerun this snippet, you should see a different version of the sample image due to random transformations. Next, we showed a sample image from `val_ds`. There is no random transformation for the validation dataset, so you should see the same image if rerunning this snippet.



Always visualize a few sample images and labels from your training and validation datasets and make sure that they look correct.

In *step 5*, we defined two data loaders for training and validation datasets. Data loaders automate the fetching of mini-batches from the training and validation datasets during training and evaluation. Creating PyTorch data loaders is simple. Simply pass the dataset and define the batch size. The batch size for the training dataset is considered a hyperparameter. So, you may want to try different values for optimum performance. For the validation dataset, the batch size does not have any impact on the performance.

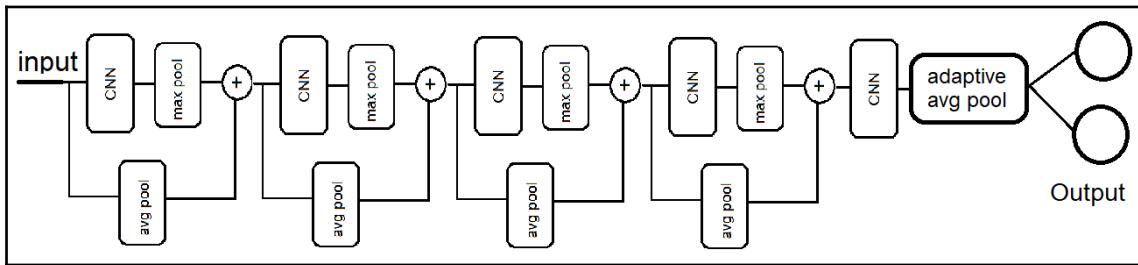
In *step 6* and *step 7*, we got a batch of data from `train_dl` and `val_dl`. As has been observed, the label batch is returned as a *list*. We needed a tensor of the `torch.float32` type. Therefore, we converted the list to a tensor and changed its type as desired. Notice that we get a batch of 8 samples from `train_dl` and a batch of 16 samples from `val_dl` based on the pre-set batch size.



After defining data loaders, always check to see that the returned batch is in the appropriate shape. You can do this simply by getting a batch from the data loaders.

Creating the model

In this recipe, we will build a model for our single-object detection problem. In our problem, we are interested in predicting the fovea center as x and y coordinates in an eye image. We will build a model consisting of several convolutional and pooling layers for this task, as shown in the following diagram:



The model will receive a resized RGB image and provide two linear outputs corresponding to the fovea coordinates. If you are interested in predicting the width or height of a bounding box for other problems, you can simply increase the number of outputs to four. In our model, we will utilize the skip connection technique introduced in the so-called **ResNet** paper. You can access the paper, *Deep Residual Learning for Image Recognition*, from <https://arxiv.org/abs/1512.03385>.

How to do it...

We will define and print the model.

1. Let's implement the model class. First, we will load the packages:

```
import torch.nn as nn
import torch.nn.functional as F
```

Then, we will define the bulk of the model class:

```
class Net(nn.Module):
    def __init__(self, params):
        super(Net, self).__init__()
    def forward(self, x):
        return x
```

Next, we will define the `__init__` function:

```
def __init__(self, params):
    super(Net, self).__init__()

    C_in,H_in,W_in=params["input_shape"]
    init_f=params["initial_filters"]
    num_outputs=params["num_outputs"]

    self.conv1 = nn.Conv2d(C_in, init_f,
kernel_size=3,stride=2,padding=1)
    self.conv2 = nn.Conv2d(init_f+C_in, 2*init_f,
kernel_size=3,stride=1,padding=1)
    self.conv3 = nn.Conv2d(3*init_f+C_in, 4*init_f,
kernel_size=3,padding=1)
    self.conv4 = nn.Conv2d(7*init_f+C_in, 8*init_f,
kernel_size=3,padding=1)
    self.conv5 = nn.Conv2d(15*init_f+C_in, 16*init_f,
kernel_size=3,padding=1)
    self.fc1 = nn.Linear(16*init_f, num_outputs)
```

Then, we will define the `forward` function:

```
def forward(self, x):
    identity=F.avg_pool2d(x,4,4)
    x = F.relu(self.conv1(x))
    x = F.max_pool2d(x, 2, 2)
    x = torch.cat((x, identity), dim=1)

    identity=F.avg_pool2d(x,2,2)
    x = F.relu(self.conv2(x))
    x = F.max_pool2d(x, 2, 2)
    x = torch.cat((x, identity), dim=1)

    identity=F.avg_pool2d(x,2,2)
    x = F.relu(self.conv3(x))
    x = F.max_pool2d(x, 2, 2)
    x = torch.cat((x, identity), dim=1)
```

The function continues like so:

```
identity=F.avg_pool2d(x,2,2)
x = F.relu(self.conv4(x))
x = F.max_pool2d(x, 2, 2)
x = torch.cat((x, identity), dim=1)

x = F.relu(self.conv5(x))
```

```
x=F.adaptive_avg_pool2d(x,1)
x = x.reshape(x.size(0), -1)

x = self.fc1(x)
return x
```

Then, we will override the Net class functions:

```
Net.__init__=__init__
Net.forward=forward
```

2. Let's now define an object of the Net class:

```
params_model={
    "input_shape": (3,256,256),
    "initial_filters": 16,
    "num_outputs": 2,
}

model = Net(params_model)
```

Then, we will move the model to the **Compute Unified Device Architecture (CUDA)** device:

```
if torch.cuda.is_available():
    device = torch.device("cuda")
    model=model.to(device)
```

Then, we will print the model:

```
print(model)
```

The model is printed in the following code block:

```
Net(
    (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1))
    (conv2): Conv2d(19, 32, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (conv3): Conv2d(51, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (conv4): Conv2d(115, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (conv5): Conv2d(243, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (fc1): Linear(in_features=256, out_features=2, bias=True)
)
```

In the next section, we will explain each step in detail.

How it works...

In *step 1*, we defined the `Net` class that outlines the model. We presented the code as a few snippets for clarity. First, we loaded the required packages. Then, we defined the class with two main functions: `__init__` and `forward`. Next, we defined the building blocks of the model inside the `__init__` function. It has five `nn.Conv2d` blocks. Notice that we set `padding=1` in all `nn.Conv2d` blocks to zero-pad both sides of the input, and keep the output size divisible by two. This is important when building networks with skip connections.



Using the `padding` parameter, you can pad both sides of the input with zeros and adjust the output of the `nn.Conv2d` layer.



Make sure that the number of input channels is correctly set when building skip connections.

Also, check out the number of input channels for each `nn.Conv2d` layer. The number of input channels, in the case of skip connections, will be the sum of the output channels from the previous layer and the skip layer.



When concatenating two tensors, they must have the same shape except in the concatenating dimension. As a reminder, here, the tensor shape is $B*C*H*W$.

There are four skip-connection blocks in the `forward` function. The last **convolutional neural network (CNN)** block is a regular `nn.Conv2d` layer without a skip connection. The output of the last CNN layer is usually referred to as **extracted features**. Then, we employed an `adaptive_avg_pool2d` layer to perform adaptive average pooling on the extracted features to get an output size of $1*1$.



The output size of `adaptive_avg_pool2d` for any input size will be whatever you specify in its argument.

Then, we reshaped or flattened the features and passed them to the linear layer. Since we are predicting the coordinate values, no activation is required for the final layer. In the end, we overrode the `__init__` and `forward` functions of the `Net` class.

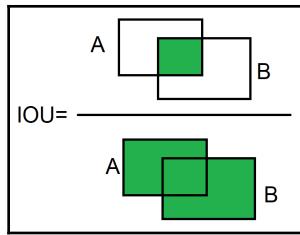
In *step 2*, we defined the model parameters and created an object of the `Net` class. We decided to downsize images to 256×256 . You can try different values, such as 128×128 or 512×512 . Also, `initial_filters`, the number of filters in the first CNN layer, was set to 16. You can try 8 or other values. The number of outputs was set to 2 since we only predict two coordinates. You can reuse this model for another single-object detection problem with four outputs to predict the width and height of objects, too.

Next, we moved the model to the CUDA device. Finally, we printed the model. Note that `print(model)` does not show functional layers created using `torch.nn.functional`.

Defining the loss, optimizer, and IOU metric

In this recipe, we will first define a loss function for our single-object detection problem. The common loss functions for detection tasks are the **mean square error (MSE)** and **smoothed-L1 loss**. Smoothed-L1 loss uses a squared term if the absolute element-wise error falls below 1, and an $L1$ term otherwise. It is less sensitive to outliers than the *MSE* and, in some cases, prevents exploding gradients. We will use the **smoothed-L1 loss**. For more details on the smoothed-L1 loss, visit <https://pytorch.org/docs/stable/nn.html#smoothl1loss>.

Then, we will define the optimizer to automatically update the model parameters. Finally, we will define a performance metric for our object detection called the **Jaccard index**, or **Intersection over Union (IOU)**. A graphical representation of IOU is shown in the following diagram:



In the next section, we will show you how to define the loss, optimizer, and IOU metric.

How to do it...

We will define the loss function, optimizer, and the IOU metric.

1. First, we will define the loss function:

```
loss_func = nn.SmoothL1Loss(reduction="sum")
```

Let's try out the loss with known values:

```
n, c=8, 2
y = 0.5 * torch.ones(n, c, requires_grad=True)
print(y.shape)

target = torch.zeros(n, c, requires_grad=False)
print(target.shape)

loss = loss_func(y, target)
print(loss.item())

y = 2 * torch.ones(n, c, requires_grad=True)
target = torch.zeros(n, c, requires_grad=False)
loss = loss_func(y, target)
print(loss.item())
```

This will print out the following results:

```
torch.Size([8, 2])
torch.Size([8, 2])
2.0
24.0
```

2. Next, we will define the optimizer:

```
from torch import optim
opt = optim.Adam(model.parameters(), lr=3e-4)
```

Then, we will define a helper function to read the learning rate:

```
def get_lr(opt):
    for param_group in opt.param_groups:
        return param_group['lr']

current_lr=get_lr(opt)
print('current lr={}'.format(current_lr))
```

This will print out the learning rate:

```
current lr=0.0003
```

3. Next, we will define a learning rate schedule:

```
from torch.optim.lr_scheduler import ReduceLROnPlateau
lr_scheduler = ReduceLROnPlateau(opt, mode='min', factor=0.5,
                                patience=20, verbose=1)
```

Let's try it out:

```
for i in range(100):
    lr_scheduler.step(1)
```

This will print out the following results:

```
Epoch      21: reducing learning rate of group 0 to 1.5000e-04.
Epoch      42: reducing learning rate of group 0 to 7.5000e-05.
Epoch      63: reducing learning rate of group 0 to 3.7500e-05.
Epoch      84: reducing learning rate of group 0 to 1.8750e-05.
```

4. Next, we will define a function to compute the IOU for a batch of data:

First, we will define a helper function to convert coordinates to a bounding box:

```
def cxcy2bbox(cxcy,w=50./256,h=50./256):
    w_tensor=torch.ones(cxcy.shape[0],1,device=cxcy.device)*w
    h_tensor=torch.ones(cxcy.shape[0],1,device=cxcy.device)*h
    cx=cxcy[:,0].unsqueeze(1)
    cy=cxcy[:,1].unsqueeze(1)
    boxes=torch.cat((cx,cy, w_tensor, h_tensor), -1) # cx, cy, w, h
    return torch.cat((boxes[:, :2] - boxes[:, 2:]/2, # xmin, ymin
                     boxes[:, :2] + boxes[:, 2:]/2), 1) # xmax, ymax
```

Let's try out the function:

```
torch.manual_seed(0)

cxcy=torch.rand(1,2)
print("center:", cxcy*256)

bb=cxcy2bbox(cxcy)
print("bounding box", bb*256)
```

This should print the following:

```
center: tensor([[127.0417, 196.6648]])
bounding box tensor([[117.0417, 186.6648, 137.0417, 206.6648]])
```

Next, we will define the metric function:

```
import torchvision
def metrics_batch(output, target):
    output=cxcy2bbox(output)
    target=cxcy2bbox(target)
    iou=torchvision.ops.box_iou(output, target)
    return torch.diagonal(iou, 0).sum().item()
```

Let's try it out on known values:

```
n,c=8,2
target = torch.rand(n, c, device=device)
target=cxcy2bbox(target)
metrics_batch(target,target)
```

The returned value will be 8.0.

5. Next, we will define the loss_batch function:

```
def loss_batch(loss_func, output, target, opt=None):
    loss = loss_func(output, target)
    with torch.no_grad():
        metric_b = metrics_batch(output,target)
    if opt is not None:
        opt.zero_grad()
        loss.backward()
        opt.step()

    return loss.item(), metric_b
```

Let's now try the `loss_batch` function on known values:

```
for xb,label_b in train_dl:  
    label_b=torch.stack(label_b,1)  
    label_b=label_b.type(torch.float32)  
    label_b=label_b.to(device)  
  
    l,m=loss_batch(loss_func,label_b,label_b)  
    print(l,m)  
    break
```

The returned values will be 0.0 8.0.

How it works...

In *step 1*, we first defined the smoothed-L1 loss function from the `torch.nn` package. Notice that we used `reduction="sum"` to return the sum of errors per mini-batch. Then, we calculated the loss value with known values. It is always beneficial to unit test the loss function using known inputs and outputs. In this case, we set the predictions to 0.5 or 1.5 values and the target values to all-zero values. For a batch size of 8 and 2 predictions, this led to 2.0 and 16.0 being printed. Later, we will use `loss.backward()` to compute the gradients of the loss with respect to the model parameters.

In *step 2*, we defined an Adam optimizer from the `torch.optim` package. The model parameters and the learning rate will be given to the optimizer. Later, we will use the `.step` method to automatically update the model parameters using this optimizer. Also, we defined a function to read the learning rate for monitoring purposes.

In *step 3*, we defined a learning rate scheduler to reduce the learning rate on plateaus. Here, we want to monitor the loss as it is decreasing, so we set `mode="min"`. We also want to wait for `patience=20` epochs on the plateau before reducing the learning rate by `factor=0.5`. We then unit tested the scheduler by fixing the monitoring metric in `.step(1)`. As expected, the learning rate is halved every 20 epochs.

In *step 4*, we developed the IOU function in a few steps. First, we defined a helper function to create a bounding box of (`width=height=20`) given the center coordinates. The helper function returns a bounding box in this format: `[x0, y0, x1, y1]`, where `x0`, `y0`, and `x1`, `y1` are the top-left and bottom-right coordinates of the bounding box, respectively. The reason for this conversion will be cleared shortly. Remember that previously, we scaled the coordinates in the range of `[0, 1]`, so we do the same for the width and height of the bounding box by dividing them by 256.

Next, we defined `metrics_batch` to compute the IOU per batch size. We used `torchvision.ops.box_iou` to compute the IOU from bounding boxes. The `box_iou` function expects the bounding box to be in $[x_0, y_0, x_1, y_1]$ format. We also test the function on known values to make sure that it works as expected. Since the batch size was set to 8, it will return 8.0 in case of a complete overlap.

In *step 5*, we defined the `loss_batch` function. This function will be used during training and evaluation. It will return the loss and IOU per batch size. Moreover, it will update the model parameters during the training phase. We also unit tested the function by passing known values to the function. As expected, for the ideal case, the loss and IOU will be 0.0 and 8.0, respectively.



Always unit test your loss function, metric function, and other helper functions with known inputs and compare the results with the expected outputs.

Training and evaluation of the model

In previous recipes, we learned to create the datasets, built the model, and defined the loss function, IOU metric, and optimizer. Now it is time to train our model using the ingredients that we prepared. For better code readability, we will define a few helper functions.

How to do it...

We will train and evaluate the model.

1. First, we will define a `loss_epoch` helper function:

```
def  
loss_epoch(model, loss_func, dataset_dl, sanity_check=False, opt=None):  
    running_loss=0.0  
    running_metric=0.0  
    len_data=len(dataset_dl.dataset)
```

The function continues with an internal loop:

```
for xb, yb in dataset_dl:  
    yb=torch.stack(yb,1)  
    yb=yb.type(torch.float32).to(device)
```

```
        output=model(xb.to(device))
        loss_b,metric_b=loss_batch(loss_func, output, yb, opt)
        running_loss+=loss_b
        if metric_b is not None:
            running_metric+=metric_b

        if sanity_check is True:
            break
```

The function ends with the following code snippet:

```
loss=running_loss/float(len_data)
metric=running_metric/float(len_data)
return loss, metric
```

2. Then, we will define a `train_val` function:

```
import copy
def train_val(model, params):
    num_epochs=params["num_epochs"]
    loss_func=params["loss_func"]
    opt=params["optimizer"]
    train_dl=params["train_dl"]
    val_dl=params["val_dl"]
    sanity_check=params["sanity_check"]
    lr_scheduler=params["lr_scheduler"]
    path2weights=params["path2weights"]
```

We continue the function by defining two Python dictionaries to keep track of the loss and metric values:

```
loss_history={
    "train": [],
    "val": []
}
metric_history={
    "train": [],
    "val": []
}
```

The function continues with a placeholder for the best model parameters and best loss:

```
best_model_wts = copy.deepcopy(model.state_dict())
best_loss=float('inf')
```

The function continues with an internal loop:

```
for epoch in range(num_epochs):
    current_lr=get_lr(opt)
    print('Epoch {}/{}, current lr={}'.format(epoch, num_epochs
- 1, current_lr))
```

The loop continues with the snippets to train the model for one epoch:

```
model.train()
train_loss,
train_metric=loss_epoch(model, loss_func, train_dl, sanity_check, opt)
loss_history["train"].append(train_loss)
metric_history["train"].append(train_metric)
```

The loop continues with the evaluation snippet:

```
model.eval()
with torch.no_grad():
    val_loss,
val_metric=loss_epoch(model, loss_func, val_dl, sanity_check)
loss_history["val"].append(val_loss)
metric_history["val"].append(val_metric)
```

The loop continues with the snippet to store the best model parameters:

```
if val_loss < best_loss:
    best_loss = val_loss
    best_model_wts = copy.deepcopy(model.state_dict())
    torch.save(model.state_dict(), path2weights)
    print("Copied best model weights!")
```

The loop continues with the learning rate schedule:

```
lr_scheduler.step(val_loss)
if current_lr != get_lr(opt):
    print("Loading best model weights!")
    model.load_state_dict(best_model_wts)
```

Within the loop, we will print the loss metric values:

```
print("train loss: %.6f, accuracy: %.2f"
%(train_loss,100*train_metric))
print("val loss: %.6f, accuracy: %.2f"
%(val_loss,100*val_metric))
print("-"*10)
```

Outside the loop, we will return the trained model, loss history, and metrics history:

```
model.load_state_dict(best_model_wts)
return model, loss_history, metric_history
```

3. Let's now train the model by calling the `train_val` function:

```
loss_func=nn.SmoothL1Loss(reduction="sum")
opt = optim.Adam(model.parameters(), lr=1e-4)
lr_scheduler = ReduceLROnPlateau(opt, mode='min', factor=0.5,
patience=20, verbose=1)

path2models= "./models/"
if not os.path.exists(path2models):
    os.mkdir(path2models)

params_train={
    "num_epochs": 100,
    "optimizer": opt,
    "loss_func": loss_func,
    "train_dl": train_dl,
    "val_dl": val_dl,
    "sanity_check": False,
    "lr_scheduler": lr_scheduler,
    "path2weights": path2models+"weights_smoothl1.pt",
}
model, loss_hist, metric_hist=train_val(model,params_train)
```

The training loop will start and you should see the progress as printed:

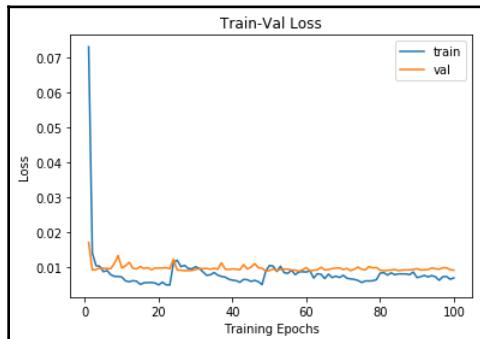
```
Epoch 0/99, current lr=0.0001
Copied best model weights!
train loss: 0.073063, accuracy: 13.08
val loss: 0.017105, accuracy: 24.20
-----
Epoch 1/99, current lr=0.0001
Copied best model weights!
train loss: 0.014088, accuracy: 31.04
val loss: 0.009260, accuracy: 52.49
```

4. Next, we will plot the training and validation loss:

```
num_epochs=params_train["num_epochs"]
plt.title("Train-Val Loss")
plt.plot(range(1,num_epochs+1),loss_hist["train"],label="train")
plt.plot(range(1,num_epochs+1),loss_hist["val"],label="val")
plt.ylabel("Loss")
```

```
plt.xlabel("Training Epochs")
plt.legend()
plt.show()
```

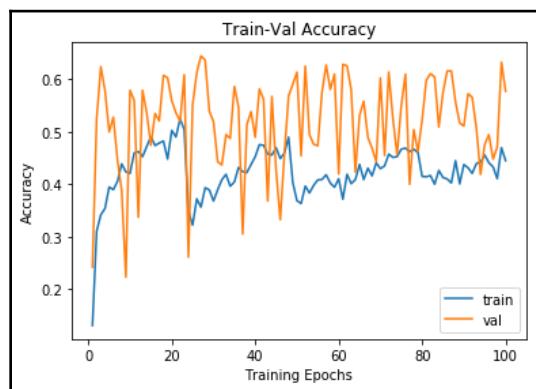
This will produce the following plot:



Next, we will plot the IOU progress:

```
plt.title("Train-Val Accuracy")
plt.plot(range(1,num_epochs+1),metric_hist["train"],label="train")
plt.plot(range(1,num_epochs+1),metric_hist["val"],label="val")
plt.ylabel("Accuracy")
plt.xlabel("Training Epochs")
plt.legend()
plt.show()
```

This will print the following screenshot:



In the next section, we will explain each step in detail.

How it works...

In *step 1*, we defined a helper function to compute the loss and IOU metric values per epoch. The function will be used for both the training and validation datasets. For the validation dataset, no optimization is performed if `opt=None` is passed to the function. In the function, batches of data are obtained from the data loader within a loop. Remember that we needed to convert the labels to tensors. Then, the model outputs are obtained and the loss and IOU were computed per mini-batches using the `loss_batch` function.

In *step 2*, we defined the `train_val` function in a few steps for better code readability. The inputs to the function are the model and the training parameters. We extracted the keys from `params`. Then, we defined `loss_history` and `metric_history` to record the loss and metric values during training. During training, we want to keep the best model parameters. Thus, we defined `best_model_wts` to store the best model parameters. To be able to track the model's performance, we need to store the best loss value. For the first training epoch, there is no previous loss value, so we initialized `best_loss` to a large number or "`inf`". Then, the main loop starts, which runs for `num_epochs` iterations. We printed the current learning rate at the beginning to keep an eye on it. Then, we set the model in training mode and trained the model. This means that the loss value is calculated and the model parameters are updated using the optimizer, with all this happening inside the `loss_epoch` function as previously described. Next, we set the model in evaluation mode and evaluated the model on the validation data. This time, no gradient calculation nor optimization were needed.



Do not forget to set the model to its proper mode as required using the `.train()` and `.eval()` methods.

After each evaluation, we compared the current validation loss with the best loss value and stored the model parameters if a better loss was observed. Next, we passed the validation loss to the learning rate schedule. If the validation loss remains unchanged for 20 epochs, the learning rate schedule will reduce the learning rate by a factor of 2. Also, we printed the progress in each epoch to monitor the training process. Finally, the loop ends and we returned the best performing model and the history of loss and metric values.



Do not forget to stop `autograd` from computing the gradients during model evaluation by wrapping the code block in `with torch.no_grad():`

In *step 3*, we used the `train_val` function to actually train the model. We created a folder to store the model parameters as a pickle file. Then, we defined the training parameters in `params_train`. You can set the `sanity_check` flag to `True` if you want to quickly execute the function and fix any possible errors. The flag breaks a training epoch after one mini-batch, meaning the loops are executed more quickly. Then, you can return to `sanity_check=False`. As has been observed, the loss and metric values were printed.

In *step 4*, we plotted the training and validation loss and IOU values. The plots show the progress of the training and evaluation.

Deploying the model

In this recipe, we will deploy the model. We will consider two deployment cases: deployment on a PyTorch dataset, and deployment on individual images stored locally. Since there is no test dataset available, we will use the validation dataset during deployment. We assume that you want to deploy the model for inference in a new script separate from the training scripts. In this case, the dataset and model do not exist in memory. To avoid repetition, we will skip defining the dataset and model in this section. Follow the instructions in the *Creating custom datasets* and *Creating the model* recipes to define the validation dataset and model in the deployment script. In the following scripts, we will assume that you have defined `Net` for the model definition and `val_ds` and `val_dl` for the validation dataset.

How to do it...

We will load the model parameters from a file and deploy the model on the validation dataset and individual images:

1. First, we will create an object of the `Net` class:

```
params_model={  
    "input_shape": (3,256,256),  
    "initial_filters": 16,  
    "num_outputs": 2,  
}  
model = Net(params_model)  
model.eval()
```

Move the model to the CUDA device:

```
if torch.cuda.is_available():
    device = torch.device("cuda")
    model=model.to(device)
```

2. Then, we will load the model parameters from the file:

```
path2weights="../models/weights.pt"
model.load_state_dict(torch.load(path2weights))
```

3. Let's verify the model by testing on the validation dataset:

```
loss_func=nn.SmoothL1Loss(reduction="sum")

with torch.no_grad():
    loss,metric=loss_epoch(model,loss_func,val_dl)
print(loss,metric)
```

This should print the following:

```
0.008903446095064282 0.5805782794952392
```

4. Next, we will deploy the model on the image samples from val_ds.

We will import the packages:

```
from PIL import ImageDraw
import numpy as np
import torchvision.transforms.functional as tv_F
np.random.seed(0)

import matplotlib.pyplot as plt
%matplotlib inline
```

We will define a function to display a tensor with two labels as bounding boxes:

```
def show_tensor_2labels(img,label1,label2,w_h=(50,50)):
    label1=rescale_label(label1,img.shape[1:])
    label2=rescale_label(label2,img.shape[1:])
    img=tv_F.to_pil_image(img)

    w,h=w_h
    cx,cy=label1
    draw = ImageDraw.Draw(img)
    draw.rectangle(((cx-w/2, cy-h/2), (cx+w/2,
    cy+h/2)),outline="green",width=2)

    cx,cy=label2
```

```
    draw.rectangle(((cx-w/2, cy-h/2), (cx+w/2,
                                         cy+h/2)), outline="red", width=2)

plt.imshow(np.asarray(img))
```

We will get random sample indices:

```
rndInds=np.random.randint(len(val_ds), size=10)
print(rndInds)
```

This should print the following indices:

```
[44 47 64 67 67  9 21 36 70 12]
```

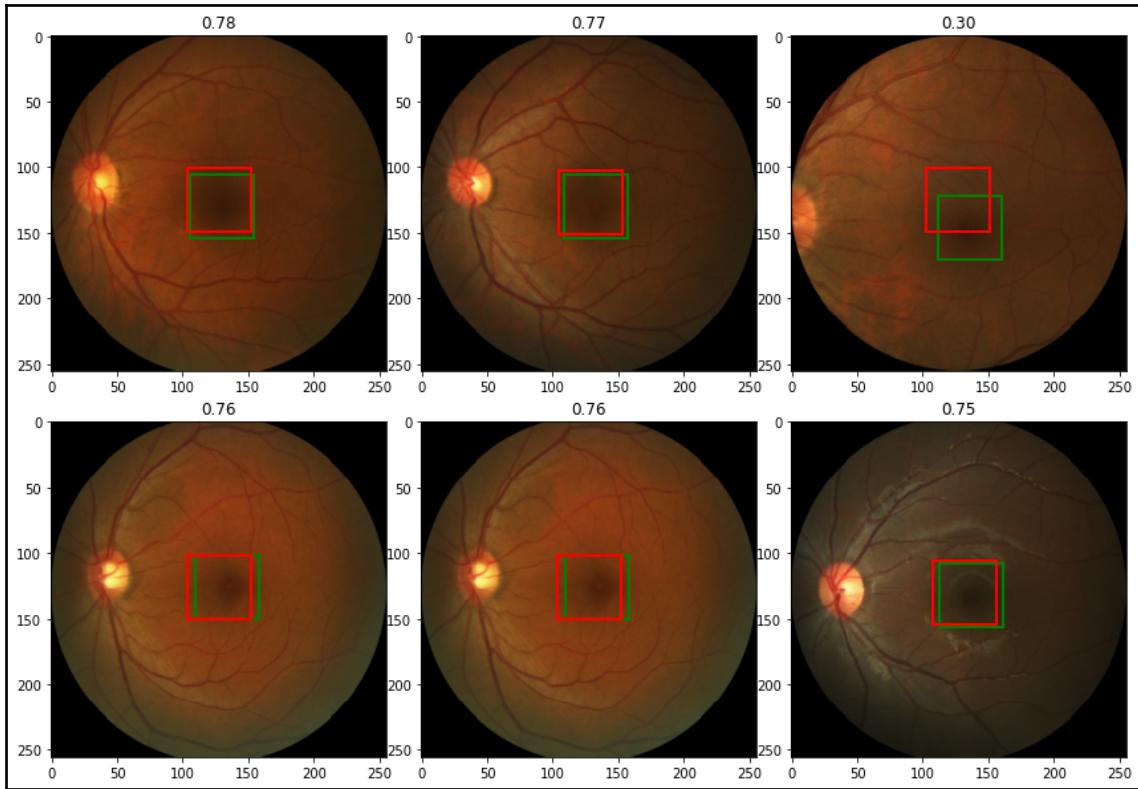
Next, we will deploy the model on the samples and display the predictions together with the ground truth:

```
plt.rcParams['figure.figsize'] = (15, 10)
plt.subplots_adjust(wspace=0.0, hspace=0.15)

for i,rndi in enumerate(rndInds):
    img,label=val_ds[rndi]
    h,w=img.shape[1:]
    with torch.no_grad():
        label_pred=model(img.unsqueeze(0).to(device))[0].cpu()
    plt.subplot(2,3,i+1)
    show_tensor_2labels(img,label,label_pred)
    # calculate IOU
    label_bb=cxcy2bbox(torch.tensor(label).unsqueeze(0))
    label_pred_bb=cxcy2bbox(label_pred.unsqueeze(0))
    iou=torchvision.ops.box_iou(label_bb, label_pred_bb)
    plt.title("%.2f" %iou.item())

    if i>4:
        break
```

The results are shown in the following screenshot:



5. Next, we will deploy the model on individual images:

```
path2labels=os.path.join(path2data,"Training400","Fovea_location.xlsx")
labels_df=pd.read_excel(path2labels,index_col="ID")

img,label=load_img_label(labels_df,1)
print(img.size, label)

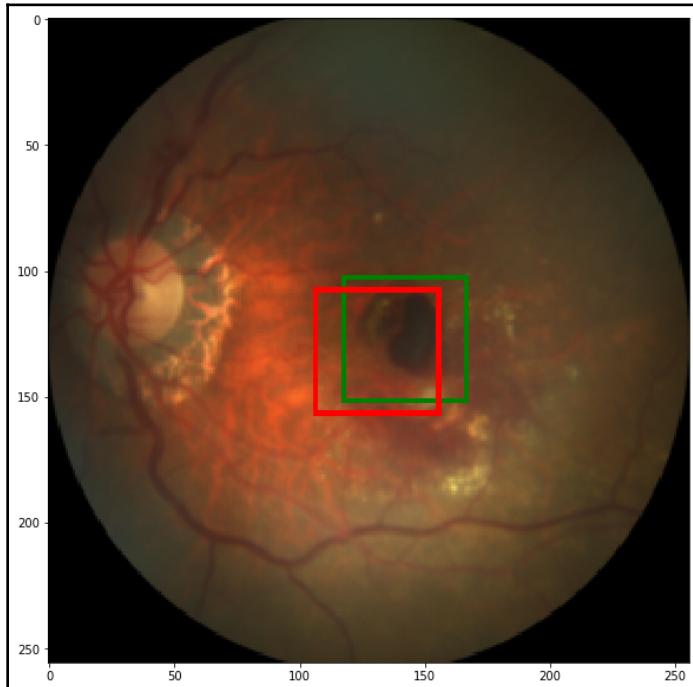
img,label=resize_img_label(img,label,target_size=(256,256))
print(img.size, label)

img=TF.to_tensor(img)
label=scale_label(label,(256,256))
print(img.shape)

with torch.no_grad():
```

```
label_pred=model(img.unsqueeze(0).to(device))[0].cpu()  
  
show_tensor_2labels(img,label,label_pred)
```

The results are shown in the following screenshot:



6. Next, we will calculate the inference time per image:

```
import time  
elapsed_times=[]  
with torch.no_grad():  
    for k in range(100):  
        start=time.time()  
        label_pred=model(img.unsqueeze(0).to(device))[0].cpu()  
        elapsed=time.time()-start  
        elapsed_times.append(elapsed)  
print("inference time per image: %.4f s" %np.mean(elapsed_times))
```

This will print the following:

```
inference time per image: 0.0014 s
```

In the next section, we will explain each step in detail.

How it works...

In *step 1*, we created an object of the `Net` class and called it `model`. We assumed that you copied the scripts of the `Net` class from the *Creating the model* recipe. Note that the model parameters are randomly initialized at this point. We then moved the model to the CUDA device for accelerated processing.

If you recall from the *Training and evaluation of the model* recipe, we stored the best model parameters as a pickle file called `weights.p`. In *step 2*, we loaded the model parameters from the pickle file into the model.

In *step 3*, we evaluated the model on the validation dataset to verify the previous steps. Oftentimes, things may go wrong during the storing and loading of model parameters. Hence, evaluating the model on the validation dataset can verify the process. Doing that, you should see the same performance that you observed during the training process. Here, we used the `loss_epoch` function, defined in the *Defining the loss, optimizer, and IOU metric* recipe, to get the loss and metrics values on the validation dataset.

In *step 4*, we deployed the model on the validation dataset. Unfortunately, we do not have a test dataset. We defined the `show_tensor_2labels` function to show the image and predictions together with the ground truth as bounding boxes. The function assumes that the image is a PyTorch tensor. We converted the tensor to a `PIL` image using the `to_pil_image` function from `torchvision`. Also, recall that the ground truth and predictions were scaled to the range of `[0, 1]`, so we rescaled them back to the image size. Then, we picked a set of random indices. Next, we get random images from `val_ds`. When you get a sample image from `val_ds`, it will be in a `3*256*256` shape, hosted on the CPU. Thus, we added a new dimension to it using `unsqueeze(0)` and moved it to CUDA before passing it to the model. The output is the predicted fovea coordinates. Next, we created bounding boxes from the coordinates using the `cxcy2bbox` function to be able to calculate the IOU metric. The calculated IOU was printed on top of each image.

You may also want to deploy the model on individual images separate from `val_ds`. This can be useful for future images that will be collected as a test dataset. We showed how to do this in *step 5*. Assuming that the new image is stored locally, we loaded it as a `PIL` image. Then, we resized the `PIL` image to `256*256`. Next, we converted the image to a PyTorch tensor using the `to_tensor` method and added a dimension to it. Finally, it was passed to the model to get the predictions.

In *step 6*, we are interested to know the inference time per image. This is an important factor during deployment. We measured the inference time by averaging the elapsed time per image for 100 iterations. As a reminder, the result shows the inference time using the GPU.

5

Multi-Object Detection

Object detection is the process of locating and classifying existing objects in an image. Identified objects are shown with bounding boxes in the image. There are two methods for general object detection: region proposal-based and regression/classification-based. In this chapter, we will use a regression/classification-based method called YOLO.

YOLO, which stands for **You Only Look Once**, is based on a series of papers that can be accessed from a variety of links (*You Only Look Once: Unified, Real-Time Object Detection* (<https://arxiv.org/abs/1506.02640>), *YOLO9000: Better, Faster, Stronger* (<https://arxiv.org/abs/1612.08242>), and *YOLOv3: An Incremental Improvement* (<https://arxiv.org/abs/1804.02767>)). YOLO-v3 is the latest version of the series and performs better in terms of accuracy than previous versions. Thus, we will focus on developing Yolo-v3 using PyTorch in this chapter.

In this chapter, we will learn how to implement the YOLO-v3 algorithm and train and deploy it for object detection using PyTorch.

In particular, we will cover the following recipes:

- Creating datasets
- Creating a YOLO-v3 model
- Defining the loss function
- Training the model
- Deploying the model

Creating datasets

We will use the COCO dataset to train the YOLO-v3 model. COCO is a large-scale object detection, segmentation, and captioning dataset. It contains 80 object categories for object detection. For more details on the COCO dataset, visit the following link: <http://cocodataset.org/#home>.

In this recipe, you will learn how to create a custom dataset, perform data transformation, and define dataloaders.

Getting ready

We will need to download the COCO dataset to complete this recipe:

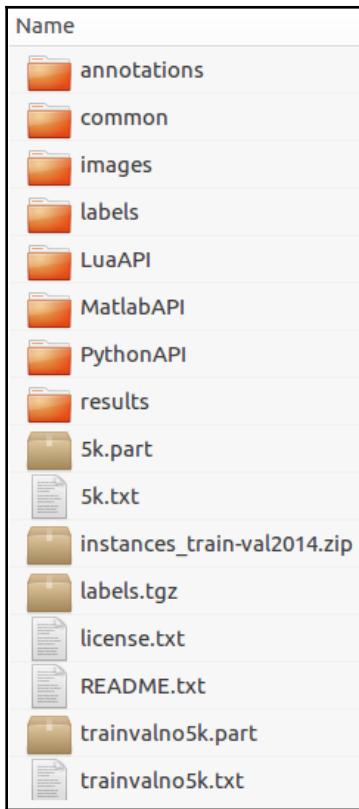
1. Download the following GitHub repository: <https://github.com/pjreddie/darknet>.
2. From the downloaded repository, get the `darknet/scripts/get_coco_dataset.sh` file.
3. Create a folder named `data` where your scripts are located and copy `get_coco_dataset.sh` into the folder.
4. Next, open a Terminal and execute the `get_coco_dataset.sh` file from the `data` folder. The script will download the complete COCO dataset into a subfolder named `coco`.



To run `get_coco_dataset.sh` on Windows, you need to install WSL. Please follow the instructions in the following link to install WSL: <https://docs.microsoft.com/en-us/windows/wsl/install-win10>.

5. Create a folder named `config` where your scripts are located and copy the `darknet/cfg/yolov3.cfg` file into the `config` folder.

A snapshot of the contents of the `coco` folder is shown in the following screenshot:



In the `images` folder, there should be two subfolders called `train2014` and `val2014` with 82783 and 40504 images, respectively.

In the `labels` folder, there should be two subfolders called `train2014` and `val2014` with 82081 and 40137 text files, respectively. These text files contain the bounding box coordinates of the objects in the images.

For instance, the `COCO_val2014_000000000133.txt` text file in the `val2014` folder contains the following coordinates:

```
59 0.510930 0.442073 0.978141 0.872188  
77 0.858305 0.073521 0.074922 0.059833
```

The first number is the object ID, while the next four numbers are normalized bounding box coordinates in [xc, yc, w, h] format, where xc, yc are the centroid coordinates and w, h are the width and height of the bounding box.

In addition, the `trainvalno5k.txt` file contains a list of 117264 images that will be used to train the model. This list is a combination of the images in the `train2014` and `val2014` subfolders, except for 5000 images. The `5k.txt` file contains a list of 5000 images that will be used for validation.

Finally, get the `coco.names` file from the following link and put it in the `data` folder:
<https://github.com/pjreddie/darknet/blob/master/data/coco.names>.

The `coco.names` file contains a list of 80 object categories in the COCO dataset.

How to do it...

Now that we've downloaded the COCO dataset, we will create training and validation datasets and dataloaders using PyTorch's `Dataset` and `Dataloader` classes.

Creating a custom COCO dataset

In this section, we will define the `CocoDataset` class and show some sample images from the training and validation datasets. Let's get started:

1. First, we will define a custom dataset class for COCO.

Import the required packages:

```
from torch.utils.data import Dataset
from PIL import Image
import torchvision.transforms.functional as TF
import os
import numpy as np
import torch

device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")
```

Define the CocoDataset class:

```
class CocoDataset(Dataset):
    def __init__(self, path2listFile, transform=None,
trans_params=None):
        with open(path2listFile, "r") as file:
            self.path2imgs = file.readlines()
        self.path2labels = [
            path.replace("images", "labels").replace(".png",
".txt").replace(".jpg", ".txt")
            for path in self.path2imgs]

        self.trans_params = trans_params
        self.transform = transform

    def __len__(self):
        return len(self.path2imgs)
```

CocoDataset continues with the following code:

```
def __getitem__(self, index):
    path2img = self.path2imgs[index % len(self.path2imgs)].rstrip()

    img = Image.open(path2img).convert('RGB')

    path2label = self.path2labels[index % len(self.path2imgs)].rstrip()

    labels= None
    if os.path.exists(path2label):
        labels = np.loadtxt(path2label).reshape(-1, 5)
    if self.transform:
        img, labels = self.transform(img, labels,
self.trans_params)

    return img, labels, path2img
```

2. Next, we will create an object of the CocoDataset class for the training data:

```
root_data=".data/coco"
path2trainList=os.path.join(root_data, "trainvalno5k.txt")
coco_train = CocoDataset(path2trainList)
print(len(coco_train))
```

The preceding snippet will print the following output:

```
117264
```

Get a sample item from `coco_train`:

```
img, labels, path2img = coco_train[1]
print("image size:", img.size, type(img))
print("labels shape:", labels.shape, type(labels))
print("labels \n", labels)
```

The preceding snippet will print the following output:

```
image size: (640, 426) <class 'PIL.Image.Image'>
labels shape: (2, 5) <class 'numpy.ndarray'>
labels
[[23.        0.770336  0.489695  0.335891  0.697559]
 [23.        0.185977  0.901608  0.206297  0.129554]]
```

3. Next, we will create an object of the `CocoDataset` class for the validation data:

```
path2valList=os.path.join(root_data, "5k.txt")
coco_val = CocoDataset(path2valList, transform=None,
trans_params=None)
print(len(coco_val))
```

The preceding snippet will print the following output:

```
5000
```

Get a sample item from `coco_val`:

```
img, labels, path2img = coco_val[7]
print("image size:", img.size, type(img))
print("labels shape:", labels.shape, type(labels))
print("labels \n", labels)
```

The preceding snippet will print the following output:

```
image size: (640, 427) <class 'PIL.Image.Image'>
labels shape: (3, 5) <class 'numpy.ndarray'>
labels
[[20.        0.539742  0.521429  0.758641  0.957143]
 [20.        0.403469  0.470714  0.641656  0.695948]
 [20.        0.853039  0.493279  0.293922  0.982061]]
```

4. Let's display a sample image from the `coco_train` and `coco_val` datasets.

Load the required packages:

```
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image, ImageDraw, ImageFont
from torchvision.transforms.functional import to_pil_image
import random
%matplotlib inline
```

Get a list of COCO object names:

```
path2cocoNames="./data/coco.names"
fp = open(path2cocoNames, "r")
coco_names = fp.read().split("\n")[:-1]
print("number of classes: ", len(coco_names))
print(coco_names)
```

The preceding snippet will print the length and a list of objects:

```
number of classes: 80
['person', 'bicycle', 'car', 'motorbike', ...]
```

Define a `rescale_bbox` helper function to rescale normalized bounding boxes to the original image size:

```
def rescale_bbox(bb,W,H):
    x,y,w,h=bb
    return [x*W, y*H, w*W, h*H]
```

Define the `show_img_bbox` helper function to show an image with object bounding boxes:

```
COLORS = np.random.randint(0, 255, size=(80, 3), dtype="uint8")
fnt = ImageFont.truetype('Pillow/Tests/fonts/FreeMono.ttf', 16)
def show_img_bbox(img,targets):
    if torch.is_tensor(img):
        img=to_pil_image(img)
    if torch.is_tensor(targets):
        targets=targets.numpy()[:,1:]
```

The `show_img_bbox` helper function continues with the following code:

```
W, H=img.size
draw = ImageDraw.Draw(img)

for tg in targets:
    id_=int(tg[0])
    bbox=tg[1:]
    bbox=rescale_bbox(bbox,W,H)
    xc,yc,w,h=bbox
    color = [int(c) for c in COLORS[id_]]
    name=coco_names[id_]
    draw.rectangle(((xc-w/2, yc-h/2), (xc+w/2,
    yc+h/2)), outline=tuple(color), width=3)
    draw.text((xc-w/2,yc-h/2),name, font=fnt,
    fill=(255,255,255,0))
plt.imshow(np.array(img))
```

In the preceding snippet, if the font that's passed to `ImageFont.truetype` is not available on your computer, you can remove `font=fnt` from the `draw.text` command. Alternatively, you may use a more common font such as the following:

```
fnt = ImageFont.truetype('arial.ttf', 16)
```

5. Call the `show_img_bbox` helper function to show a sample image from `coco_train`:

```
np.random.seed(2)
rnd_ind=np.random.randint(len(coco_train))
img, labels, path2img = coco_train[rnd_ind]
print(img.size, labels.shape)

plt.rcParams['figure.figsize'] = (20, 10)
show_img_bbox(img,labels)
```

The preceding snippet will print the following output:

```
(640, 428) (2, 5)
```

The sample image with its object bounding boxes can be seen in the following screenshot:



As we can see, the person and skateboard were highlighted in the image using two bounding boxes.

6. Call the `show_img_bbox` helper function to show a sample image from `coco_val`:

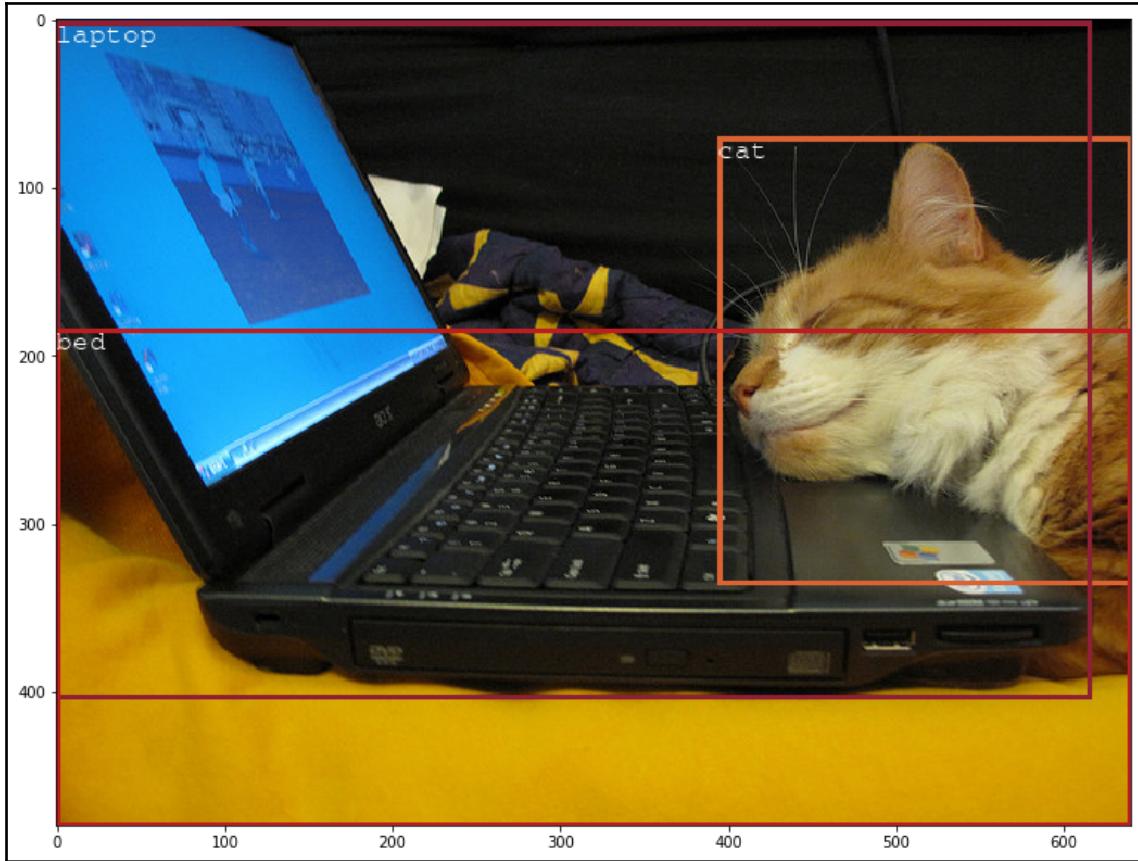
```
np.random.seed(0)
rnd_ind=np.random.randint(len(coco_val))
img, labels, path2img = coco_val[rnd_ind]
print(img.size, labels.shape)

plt.rcParams['figure.figsize'] = (20, 10)
show_img_bbox(img,labels)
```

The preceding snippet will print the following output:

```
(640, 480) (3, 5)
```

The sample image with its object bounding boxes can be seen in the following screenshot:



In the next section, we will develop transformation functions.

Transforming data

In this section, we will define a `transform` function and the parameters to be passed to the `CocoDataset` class. Let's get started:

1. First, we will define a `pad_to_square` helper function:

```
def pad_to_square(img, boxes, pad_value=0, normalized_labels=True):
    w, h = img.size
    w_factor, h_factor = (w,h) if normalized_labels else (1, 1)
    dim_diff = np.abs(h - w)
    pad1= dim_diff // 2
    pad2= dim_diff - pad1
```

The `pad_to_square` helper function continues with the following code:

```
if h<=w:
    left, top, right, bottom= 0, pad1, 0, pad2
else:
    left, top, right, bottom= pad1, 0, pad2, 0
padding= (left, top, right, bottom)

img_padded = TF.pad(img, padding=padding, fill=pad_value)
w_padded, h_padded = img_padded.size
x1 = w_factor * (boxes[:, 1] - boxes[:, 3] / 2)
y1 = h_factor * (boxes[:, 2] - boxes[:, 4] / 2)
x2 = w_factor * (boxes[:, 1] + boxes[:, 3] / 2)
y2 = h_factor * (boxes[:, 2] + boxes[:, 4] / 2)
```

The `pad_to_square` function continues with the following code:

```
x1 += padding[0] # left
y1 += padding[1] # top
x2 += padding[2] # right
y2 += padding[3] # bottom
boxes[:, 1] = ((x1 + x2) / 2) / w_padded
boxes[:, 2] = ((y1 + y2) / 2) / h_padded
boxes[:, 3] *= w_factor / w_padded
boxes[:, 4] *= h_factor / h_padded

return img_padded, boxes
```

2. Define the `hflip` helper function to horizontally flip images:

```
def hflip(image, labels):
    image = TF.hflip(image)
    labels[:, 1] = 1.0 - labels[:, 1]
    return image, labels
```

3. Define the transformer function:

```
def transformer(image, labels, params):
    if params["pad2square"] is True:
        image, labels= pad_to_square(image, labels)
    image = TF.resize(image,params["target_size"])

    if random.random() < params["p_hflip"]:
        image,labels=hflip(image,labels)

    image=TF.to_tensor(image)
    targets = torch.zeros((len(labels), 6))
    targets[:, 1:] = torch.from_numpy(labels)
    return image, targets
```

4. Now, let's create an object of CocoDataset for training data by passing the transformer function:

```
trans_params_train={
    "target_size" : (416, 416),
    "pad2square": True,
    "p_hflip" : 1.0,
    "normalized_labels": True,
}
coco_train= CocoDataset(path2trainList,
                        transform=transformer,
                        trans_params=trans_params_train)
```

Display a sample image from coco_train:

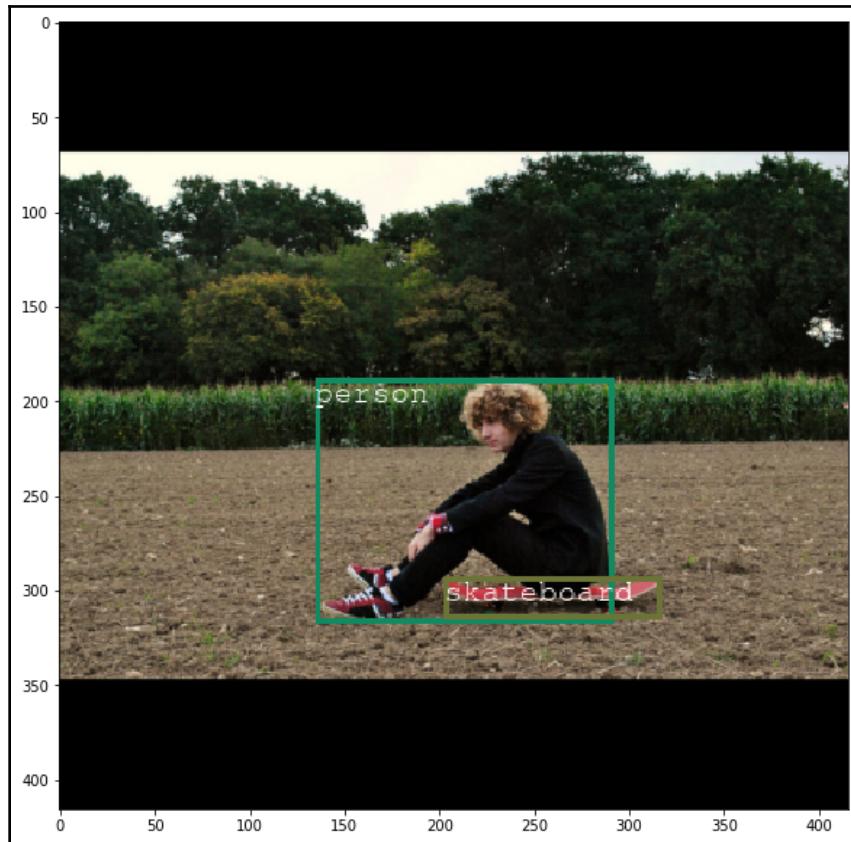
```
np.random.seed(2)
rnd_ind=np.random.randint(len(coco_train))
img, targets, path2img = coco_train[rnd_ind]
print("image shape:", img.shape)
print("labels shape:", targets.shape)

plt.rcParams['figure.figsize'] = (20, 10)
COLORS = np.random.randint(0, 255, size=(80, 3), dtype="uint8")
show_img_bbox(img,targets)
```

The preceding snippet will print the following output:

```
image shape: torch.Size([3, 416, 416])
labels shape: torch.Size([2, 6])
```

The sample image can be seen in the following screenshot:



As we can see, the person and skateboard were horizontally flipped in the image.

5. Similarly, we will define an object of `CocoDataset` by passing the `transformer` function to validate the data:

```
trans_params_val={  
    "target_size" : (416, 416),  
    "pad2square": True,  
    "p_hflip" : 0.0,  
    "normalized_labels": True,  
}  
coco_val= CocoDataset(path2valList,  
                      transform=transformer,  
                      trans_params=trans_params_val)
```

We will display a sample image from `coco_val`:

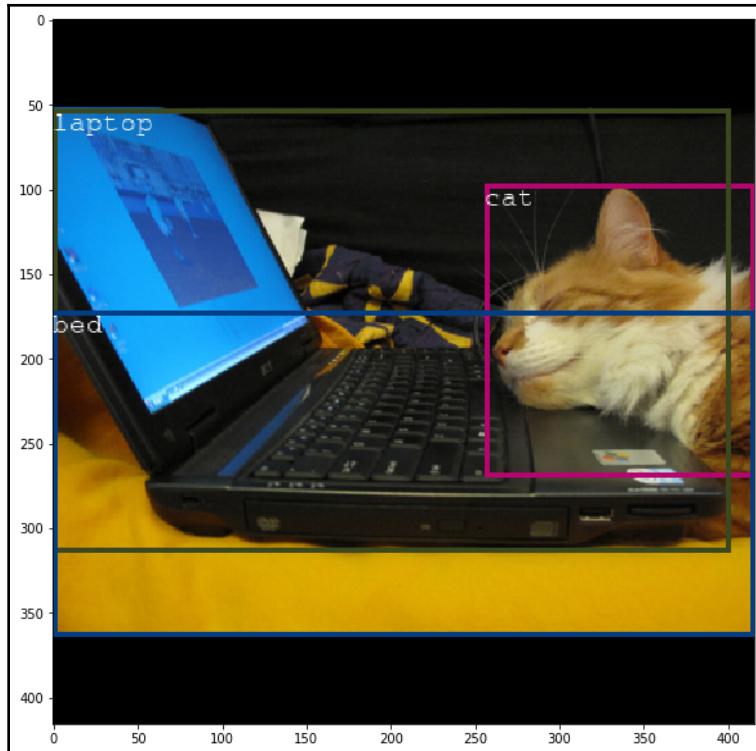
```
np.random.seed(0)
rnd_ind=np.random.randint(len(coco_val))
img, targets, path2img = coco_val[rnd_ind]
print("image shape:", img.shape)
print("labels shape:", targets.shape)

plt.rcParams['figure.figsize'] = (20, 10)
COLORS = np.random.randint(0, 255, size=(80, 3), dtype="uint8")
show_img_bbox(img,targets)
```

The preceding snippet will print the following output:

```
image shape: torch.Size([3, 416, 416])
labels shape: torch.Size([3, 6])
```

The sample image can be seen in the following screenshot:



In the next section, we will define dataloaders.

Defining the Dataloaders

In this section, we will define two dataloaders for training and validation of datasets so we can get mini-batches of data from `coco_train` and `coco_val`. Let's get started:

1. Define an object of the `Dataloader` class for the training data:

```
from torch.utils.data import DataLoader

batch_size=8
train_dl = DataLoader(
    coco_train,
    batch_size=batch_size,
    shuffle=True,
    num_workers=0,
    pin_memory=True,
    collate_fn=collate_fn,
)
```

Here, `collate_fn` is defined as follows:

```
def collate_fn(batch):
    imgs, targets, paths = list(zip(*batch))
    # Remove empty boxes
    targets = [boxes for boxes in targets if boxes is not None]
    # set the sample index
    for b_i, boxes in enumerate(targets):
        boxes[:, 0] = b_i
    targets = torch.cat(targets, 0)
    imgs = torch.stack([img for img in imgs])
    return imgs, targets, paths
```

Let's extract a mini-batch from `train_dl`:

```
torch.manual_seed(0)
for imgs_batch,tg_batch,path_batch in train_dl:
    break
print(imgs_batch.shape)
print(tg_batch.shape,tg_batch.dtype)
```

The preceding snippet will print the following output:

```
torch.Size([8, 3, 416, 416])
torch.Size([32, 6]) torch.float32
```

2. Define an object of the `Dataloader` class for the validation data:

```
val_dl = DataLoader(  
    coco_val,  
    batch_size=batch_size,  
    shuffle=False,  
    num_workers=0,  
    pin_memory=True,  
    collate_fn=collate_fn,  
)
```

Let's extract a mini-batch from `val_dl`:

```
torch.manual_seed(0)  
for imgs_batch,tg_batch,path_batch in val_dl:  
    break  
print(imgs_batch.shape)  
print(tg_batch.shape,tg_batch.dtype)
```

The preceding snippet will print the following output:

```
torch.Size([8, 3, 416, 416])  
torch.Size([83, 6]) torch.float32
```

In the next section, we will explain how each step works.

How it works...

In the *Creating a custom COCO dataset* subsection, we created a PyTorch dataset class for the COCO dataset. First, we imported the required packages. Then, we defined the `CocoDataset` class. The class started by defining the `__init__` function with three inputs:

- `path2listFile`: A string stating the location of the text file that contains a list of images
- `transform`: A function that defines various transformations, such as resizing and converting into tensors
- `trans_params`: A Python dictionary that defines transformation parameters such as the target image size

In the `__init__` function, we read the text file and loaded the list of images into `self.path2imgs`. The file contained the full path to the images that will be loaded in the `__getitem__` function. Then, we extracted the full path of labels by replacing "images" with "labels" and ".jpg", ".png" with ".txt" in the path to the images. In the `__len__` function, we returned the length of the dataset.

Next, we defined the `__getitem__` function. This function has one input, which is the index of the image to be loaded. We obtained the full path to the image and labels from `self.path2imgs` and `self.path2labels`, respectively. Then, we loaded the image as a `PIL` object. The labels were loaded as a numpy array. Then, we applied the transformations on the image and labels. Finally, we returned the image, labels, and the full path to the image. If no transformation was applied, the returned image and labels were `PIL` and `numpy` objects.

In *step 2*, we created an object of the `CocoDataset` class using the list of training data in `"trainvalno5k.txt"`. We did not pass a transformation function. As we saw, `coco_train` contains 117264 images to be used for training. Then, we got a sample image from `coco_train` and printed the size and type of the image and labels. As expected, a `PIL` image and `numpy` array were returned. Check out the printed labels. The first number in each row was the object ID, while the next four numbers were the normalized bounding box coordinates.

In *step 3*, similar to *step 2*, we created an object of the `CocoDataset` class using the list of validation data in `"5k.txt"`. As we saw, there were 5000 images in `coco_val`. Then, we obtained a sample item from `coco_val` and printed the type and size of the image and labels. You can try to fetch different images by changing the index. As we saw, without any transformation, images have different sizes.

In *step 4*, we displayed sample images and object bounding boxes from the `coco_train` and `coco_val` datasets. First, we imported the required packages. Then, we loaded COCO object names from the `coco.names` file into a list. As expected, there were 80 object categories in the file. We will use the names to display the object name on the bounding boxes.

Next, we defined the `rescale_bbox` helper function to rescale the normalized bounding boxes to the original image size. Also, to show the bounding boxes in different colors, we defined `COLORS`, an array of random tuples.

Then, we defined the `show_img_bbox` helper function with two inputs:

- `img`: Can be a `PIL` image or a PyTorch tensor that has a shape of $3 \times H \times W$.
- `targets`: The bounding box coordinates. This can be a `numpy` array that has a shape of $n \times 5$ or a PyTorch tensor that has a shape of $n \times 6$.

In the function, we check the input data type. If the inputs are PyTorch tensors, we convert them into a `PIL` image and a `numpy` array. If `img` is a tensor, we convert it into a `PIL` image using the `to_pil_image` function from `torchvision`. If the target is a tensor, we convert it into a `numpy` array using the `.numpy()` method and skip the first index of the second dimension.

In the rest of the helper function, we looped over the bounding boxes and added them to the image. Notice that we got the bounding box color and name from `COLORS` and `names` (based on the object index), respectively. The name was put, as a piece of text, on the top left corner of the bounding box.

Next, we called the `show_img_bbox` helper function to display a sample image and its bounding boxes from `coco_train` and `coco_val`. You can comment out the `np.random.seed` line to see a different image in each rerun.

In the *Transforming the data* subsection, we defined the functions required for data transformation. These transformations were required to resize images, augment the data, or convert the data into PyTorch tensors.

We started by defining the `pad_to_square` helper function with four inputs:

- `img`: A `PIL` image
- `boxes`: A `numpy` array with a shape of `(n, 5)` that contains `n` bounding boxes
- `pad_value`: The pixel fill value, which defaults to zero
- `normalized_labels`: A flag to show whether the bounding boxes were normalized to the range `[0, 1]`

This function takes a `PIL` image and pads its borders so it become a square image. In the function, we got the image size and the scaling factors of the labels. Then, we calculated the padding size and divided it into two values: `pad1` and `pad2`. For instance, if the padding size is 100, then we have `pad1= pad2= 50`. But if the padding size is 101, then we get `pad1=50` and `pad2=51`.

Then, we calculated the padding size on each side of the image. If the height is less than the width, we only need to add pixels to the top and bottom of the image. Otherwise, we add pixels to the left and right of the image.

After padding the image, we adjusted the bounding box coordinates based on the padding size. To this end, we extracted the top-left `x1, y1` and bottom-right `x2, y2` coordinates of the bounding boxes before padding.



The boxes array is in `[id, xc, yc, w, h]` format, where `id` is the object identifier, `xc`, `yc` is the centroid coordinates, and `w`, `h` is the width and height of the bounding box.

Then, we adjusted `x1`, `y1`, `x2`, `y2` by adding the padding sizes. Next, we calculated the bounding boxes using the adjusted values of `x1`, `y1`, `x2`, `y2`. Note that we normalized the labels again to the range of `[0, 1]`. The function returned a square `PIL` image and its labels as a `numpy` array.

In *step 2*, we defined the `hflip` helper function to horizontally flip images and labels.

In *step 3*, we defined the `transformer` function with three inputs:

- `image`: A `PIL` image
- `labels`: Bounding boxes as a `numpy` array that's `(n, 5)` in size
- `params`: A Python dictionary containing the transformation parameters

The function takes a `PIL` image and its labels and returns the transformed image and labels as PyTorch tensors. In the function, we check for the `pad2square` flag and, if it's `True`, we call the `pad_to_square` function. Then, the image is resized to `416*416`, which is the standard size for a YOLO-v3 network. Next, we called the `hflip` function to randomly flip the image for data augmentation. Finally, we converted the `PIL` image into a PyTorch tensor using the `to_tensor` function from `torchvision`. The labels were also converted into a PyTorch tensor of size `n*6`. The extra dimension will be used to index images in a mini-batch.

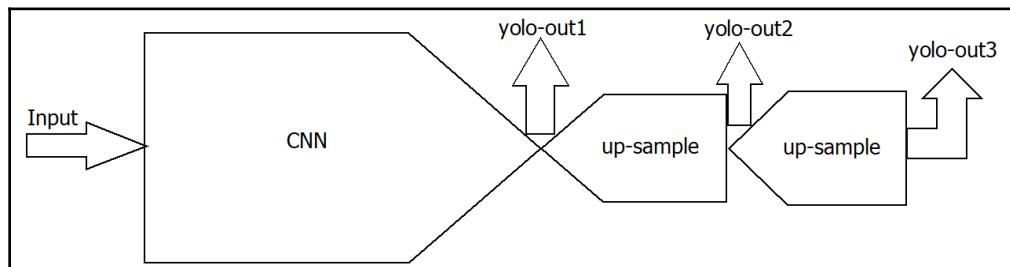
In *step 4*, we redefined `coco_train`; however, this time, we passed `transformer` and `trans_params_train` to the `CocoDataset` class. To force the horizontal flip, we set the `p_hflip` probability to `1.0`. In practice, we usually set the probability to `0.5`. You can see the effect of the transformations on the sample image. The image has been zero-padded from the top and bottom, resized to `416*416`, and horizontally flipped.

Similarly, in *step 5*, we redefined `coco_val`. We did not need data augmentation for the validation data, so we set the probability of `p_hflip` to `0.0`. Check out the transformed sample size. It has been zero-padded from the top and bottom and resized to `416*416` but not flipped.

In *Defining the Dataloaders* subsection, we defined two objects of the Dataloader class, `train_dl` and `val_dl`, for the training and validation datasets, respectively. We also defined the `collate_fn` function to process a mini-batch and return PyTorch tensors. The function was given as an argument to the Dataloader class so that the process happens on the fly. In the function, we grouped the images, targets, and paths in the mini-batch using `zip(*iterable)`. Then, we removed any empty bounding boxes in the targets. Next, we set the sample index in the mini-batch. Finally, we concatenated the images and targets as PyTorch tensors. To see how this works, we extracted a mini-batch from `train_dl` and `val_dl` and printed the shape of the returned tensors. Notice that the batch size was set to 16 in `train_dl`, whereas it was set to 32 in `val_dl`. Also, there are 87 bounding boxes in the mini-batch of `train_dl` but there are 250 in `val_dl`.

Creating a YOLO-v3 model

The YOLO-v3 network is built of convolutional layers with stride 2, skip connections, and up-sampling layers. There are no pooling layers. The network receives an image whose size is 416×416 as input and provides three YOLO outputs, as shown in the following figure:



The network down-samples the input image by a factor of 32 to a feature map of size 13×13 , where `yolo-out1` is provided. To improve the detection performance, the 13×13 feature map is up-sampled to 26×26 and 52×52 , where we have `yolo-out2` and `yolo-out3`, respectively. A cell in a feature map predicts three bounding boxes that correspond to three predefined anchors. As a result, the network predicts $13 \times 13 \times 3 + 26 \times 26 \times 3 + 52 \times 52 \times 3 = 10647$ bounding boxes in total.

A bounding box is defined using 85 numbers:

- Four coordinates, [x, y, w, h]
- An abjectness score
- C=80 class predictions corresponding to 80 object categories in the COCO dataset

In this recipe, we will show you how to develop a YOLO-v3 model using PyTorch.

How to do it...

In this recipe, we will define a few helper functions to parse the configuration file, create PyTorch modules, and define the Darknet model.

Parsing the configuration file

We need to parse the configuration file to be able to build the model. We have provided a `myutils.py` file that contains a helper function with which you can do this. The configuration file `yolov3.cfg` was downloaded previously in the Creating Datasets: Getting Ready recipe. Let's get started:

1. First, we will import the `parse_model_config` helper function:

```
from myutils import parse_model_config
```

2. Let's read and print out the config file using the `parse_model_config` helper function:

```
path2config="../config/yolov3.cfg"
blocks_list=parse_model_config(path2config)
blocks_list
```

The preceding snippet will print the following output:

```
[{'type': 'net',
  'batch': '1',
  'subdivisions': '1',
  'width': '416',
  'height': '416',
  ...
}
```

In the next section, we will create PyTorch modules for the YOLO-v3 algorithm.

Creating PyTorch modules

In this section, we will create PyTorch modules based on our parsed configuration file. We have provided the `myutils.py` file, which contains a helper function you can use to create PyTorch modules for the YOLOv3 network. Let's get started:

1. First, we will import the `create_layers` helper function to convert `blocks_list` into PyTorch modules:

```
from myutils import create_layers
```

2. Now, let's call the `create_layers` function and get a list of PyTorch modules:

```
hy_pa, m_l=create_layers(blocks_list)
print(hy_pa)
print(m_l)
```

This will print out the necessary hyperparameters:

```
{'type': 'net',
'batch': '1',
'subdivisions': '1',
'width': '416',
'height': '416',
'channels': '3',  
....
```

A list of modules will be also printed:

```
ModuleList(
(0): Sequential(
(conv_0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
(batch_norm_0): BatchNorm2d(32, eps=1e-05, momentum=0.9, affine=True,
track_running_stats=True)
(leaky_0): LeakyReLU(negative_slope=0.1)
)
....
```

In the next section, we will define the Darknet model.

Defining the Darknet model

Now, let's learn how to define the Darknet class:

1. We start by defining the `__init__` function:

```
class Darknet(nn.Module):
    def __init__(self, config_path, img_size=416):
        super(Darknet, self).__init__()
        self.blocks_list = parse_model_config(config_path)
        self.hyperparams, self.module_list =
create_layers(self.blocks_list)
        self.img_size = img_size
```

2. Next, we define the `forward` function of the Darknet class:

```
def forward(self, x):
    img_dim = x.shape[2]
    layer_outputs, yolo_outputs = [], []
    for block, module in zip(self.blocks_list[1:], self.module_list):
        if block["type"] in ["convolutional", "upsample",
"maxpool"]:
            x = module(x)
```

The `forward` function continues with the following code:

```
elif block["type"] == "shortcut":
    layer_ind = int(block["from"])
    x = layer_outputs[-1] + layer_outputs[layer_ind]
elif block["type"] == "yolo":
    x= module[0](x)
    yolo_outputs.append(x)
elif block["type"] == "route":
    x = torch.cat([layer_outputs[int(l_i)] for l_i in block["layers"].split(",")], 1)
    layer_outputs.append(x)
yolo_out_cat = torch.cat(yolo_outputs, 1)
return yolo_out_cat, yolo_outputs
```

3. Let's create an object of the Darknet class:

```
model = Darknet(path2config).to(device)
print(model)
```

The preceding snippet will print the following output:

```
Darknet(
    (module_list): ModuleList(
        (0): Sequential(
            (conv_0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
            (batch_norm_0): BatchNorm2d(32, eps=1e-05, momentum=0.9,
affine=True, track_running_stats=True)
            (leaky_0): LeakyReLU(negative_slope=0.1)

    ...
)
```

4. Next, let's test the model using a dummy input:

```
dummy_img=torch.rand(1,3,416,416).to(device)
with torch.no_grad():
    dummy_out_cat, dummy_out=model.forward(dummy_img)
    print(dummy_out_cat.shape)
    print(dummy_out[0].shape,dummy_out[1].shape,dummy_out[2].shape)
```

The preceding snippet will print the following output:

```
torch.Size([1, 10647, 85])
torch.Size([1, 507, 85]) torch.Size([1, 2028, 85]) torch.Size([1,
8112, 85])
```

In the next section, we will explain how each step works in detail.

How it works...

In the *Parsing the configuration file* subsection, we defined a helper function to parse the YOLO-v3 configuration file. For brevity, we defined the helper function in `myutils.py`, which is provided as part of this book. The input to the function is `path2file`, a string, that can be located at `"./config/yolov3.cfg"`.

The function parses the configuration file and returns the layers as a list of dictionaries. In the helper function, we split the file into lines using `.split("\n")`. Then, we removed any empty lines and comments. Next, we removed white spaces leading or tailoring a line using `rstrip()` and `lstrip()` respectively. Next, we defined an empty list called `layers_list` and looped over the lines. If a line starts with "[", a new layer is created. For a new layer, we append a dictionary to `layers_list`. The dictionary has a "type" key, which shows the type of the layer. The first layer's type is "net", which defines the hyperparameters of the network. We verify that the "height" and "width" parameters are set to "416". If a line does not start with "[", it is added to the current dictionary as an attribute of the layer.

In the *Creating PyTorch modules* subsection, we defined a helper function called `create_layers` to create PyTorch modules corresponding to the layers. For brevity, we defined the helper function in the `myutils.py` file and imported the helper function from `myutils.py`. Nevertheless, we will explain the helper function.

The input to the function is `blocks_list`, which provides a list of layers that were parsed from the configuration file in the *Parsing the configuration file* subsection.

In the function, we extracted the hyperparameters from the first block. Next, we created `channels_list` to hold the number of input channels for each layer. For the first layer, the number of input channels is 3 (the same as the image channels). Next, we created an object of the `nn.ModuleList` class to hold the submodules. Then, we created a loop to go over the list of the blocks.

As a reminder, the first block contains the model's hyperparameters.



Next, we created an object of the `nn.Sequential` class to add multiple layers to a module. For instance, for "convolutional" blocks, we sequentially added `nn.Conv2d`, `nn.BatchNorm2d`, and `nn.LeakyReLU` using the `add_module` method.

Next, in the main loop, we checked for up-sampling blocks. Up-sampling layers were used in YOLO-v3 to up-sample the feature maps to a higher resolution so that they could be concatenated with earlier layers. The `nn.Upsample` module takes the `stride=2` parameter and up-samples the feature maps by a factor of 2. We used the default `mode="nearest"` for up-sampling.

Next, in the loop, we checked for shortcut blocks. They are used to create skip connections.

A shortcut block in the configuration file was defined as follows:



```
[shortcut]
from=-3
activation=linear
```

Here, `from=-3` means that the skip connection comes from the third layer backward from the shortcut block.

The number of filters was set to that of the skip connection stored in `channels_list`.

Next, in the loop, we checked for route blocks.



The route blocks were defined with one or two arguments in the configuration file.

If defined with one argument, as shown in the following snippet, the feature maps of the fourth layer, backward from the route layer, were returned:

```
[route]
layers = -4
```

If defined with two arguments, they were used to concatenate feature maps from multiple layers. For example, in the following snippet, the feature maps from the previous layer and the 61st layer were concatenated along the depth dimension:

```
[route]
layers = -1, 61
```

Note that the actual concatenation will happen in the `forward` function of the network class. Here, we only summed the number of filters in the route layers.

Next in the loop, we checked for `yolo` blocks. There were three `yolo` blocks that corresponded to three detection outputs in the configuration file. Let's look at the first `yolo` block of the configuration file:

```
[yolo]
mask = 6,7,8
anchors = 10,13, 16,30, 33,23, 30,61, 62,45, 59,119, 116,90, 156,198,
373,326
classes=80
num=9
jitter=.3
```

```
ignore_thresh = .5
truth_thresh = 1
random=1
```

In our code, we extracted the `anchors` attribute and grouped the blocks as a list of nine tuples corresponding to nine anchors in YOLO-v3. The first tuple is (10, 13) at index 0, while the last tuple is (373, 326) at index 8.

There are three anchors per YOLO output.



Next, we extracted the `mask` attribute as a list of three values. Since there are three anchors per `yolo` block, the `mask` attribute represents the indices of those three anchors. For instance, the last three anchors indexed at 6, 7, 8 belong to the first `yolo` output. Thus, the anchors for the first `yolo` output were as follows:

```
anchors = [(116,90), (156,198), (373,326)]
```

Then, we extracted the number of object categories and the image size, which are 80 and 416, respectively. At this point, we checked all the layer types in the main loop of the `create_layers` function. Each layer type resulted in a PyTorch module that was appended to `module_list`. Then, we returned `module_list` and `hyperparams`.

Next, we defined the `EmptyLayer` class, which does nothing in a skip or shortcut connection. Then, we defined the `YOLOLayer` class. First, we defined the bulk of the `YOLOLayer` class and added the functions one by one for code readability. Then, we defined the `__init__` function and initialized the variables. The function has three inputs:

- `anchors`, which is a list of three tuples
- `num_classes`, which is 80 categories for the COCO-2107 dataset
- `img_dim`, which is an image dimension of 416

Next in the class, we defined the `forward` function with one input. The input to the function is a feature map with a shape of [batch_size, 3*85, 13, 13] for the first `yolo` layer. The function started by extracting the batch size and grid size. For the first `yolo` layer, the grid size is 13. Next, we reshaped the input feature map to [batch_size, 3, 13, 13, 85].

Next, we applied the `sigmoid` function to the abjectness score and class predictions. As we mentioned previously, the network predicts 85 values per bounding box: four values as `[x, y, w, h]`, an abjectness score, and 80 class predictions. Then, we computed the grid offsets using the `compute_grid_offsets` helper function. We defined this helper function in *step 4*. Next, we transformed the predictions into bounding boxes using the `transform_outputs` helper function, which was defined in *step 5*.

Then, the bounding box predictions were concatenated with the abjectness scores and class predictions and returned as the output of the function.

Next, we defined the `compute_grid_offsets` helper function. The helper function is a method of the `YOLOLayer` class and has three inputs:

- `self`: This refers to the `YOLOLayer` class.
- `grid_size`: The grid size, which can be 13, 26, 52.
- The `cuda` flag: `True` or `False` value depending on the availability of CUDA/GPU devices.

For a given `grid_size=13`, the function creates a grid of 13×13 on the `x` and `y` dimensions. It also scales the anchors by `self.stride=32`.

In *step 5*, we defined the `transform_outputs` helper function. This function is a method of the `YOLOLayer` class and has two inputs:

- `self`: Refers to the `YOLOLayer` class
- `predictions`: A tensor of shape `(batch_size, 3, grid_size, grid_size, 85)`

This function applied `sigmoid` non-linearity, added grid offsets to the `x`, `y` coordinates, and scaled them to `w, h` with the scaled anchors. The output of this helper function was the scaled bounding box predictions.

In *step 2*, we called the `create_layers` function to make sure it works properly. The input to this function is the `blocks_list` that we obtained by parsing the configuration file. The function returned the hyperparameters and a list of 106 PyTorch modules corresponding to the parsed layers. Check out the `yolo` layers at 82, 94, and 106.

In the *Defining the Darknet model* subsection, we defined the `Darknet` class, which is the complete detection network. First, we defined the `__init__` function with three inputs:

- `self`: Refers to the `Darknet` class
- `config_path`: The location of the config file
- `img_size`: The image dimension size, whose default is `416`

In the `__init__` function, we called `the_parse_model_config` and `create_layers` helper functions to get a list of blocks and PyTorch modules. These are the building blocks of the YOLO-v3 network.

In *step 2*, we defined the `forward` function with two inputs:

- `self`: Refers to the `Darknet` class
- `x`: A tensor of shape `(batch_size, 3, 416, 416)`

In the `forward` function, we initially defined two lists to keep track of the outputs of each layer and those of the `yolo` layers. Then, we looped over the list of blocks and modules.



As a reminder, the first block belongs to the hyperparameters. As such, in `self.blocks_list[1:]`, the index starts at 1.

In the loop, if we encountered `convolutional` blocks, we passed the input tensor to the module and appended the output to the `layer_outputs` list. Then, if we encountered `shortcut` blocks, we extracted the shortcut layer index and added the output of the shortcut layer to that of the previous layer. The result was then appended to the `layer_outputs` list. Next, if we encountered `yolo` layers, we appended the module output to both the `layer_outputs` and `yolo_outputs` lists. Lastly, if we encountered `route` layers, we extracted the route layer indices and concatenated the outputs of the route layers. Eventually, we concatenated the tensors of the `yolo_outputs` list as a `yolo_out_cat` tensor and returned both outputs.

In *step 3*, we created an object of the `Darknet` class called `model` and printed it. As we saw, the complete YOLO-v3 architecture was created. We checked the model architecture when it was printed. It should have 106 layers with three `YOLOLayer` modules at indices 82, 94, and 106.



There are a total of 106 layers in the YOLO-v3 architecture. Layers 82, 94, 106 are the three `yolo` outputs.

In *step 4*, to check the architecture and verify our code, we passed a dummy image to the model and obtained the output. As a reminder, there were three `yolo` outputs of shape $(1, 507, 85)$, $(1, 2028, 85)$, and $(1, 8112, 85)$. The final output was the concatenation of these three layers in dimension 1. The output size was $(1, 10647, 85)$ since $507+2028+8112=10647$. In other words, for every input image, YOLO-v3 predicts 10647 bounding boxes.

Defining the loss function

In this recipe, we will define a loss function for the YOLO-v3 architecture. To get an insight into of the YOLO-v3 loss, recall that the model output comprises the following elements:

- $[x, y, w, h]$ of bounding boxes
- An objectness score
- Class predictions for 80 object categories

Thus, the YOLO-v3 loss function is composed of the following:

$$\text{loss} = \text{loss}_x + \text{loss}_y + \text{loss}_w + \text{loss}_h + \text{loss}_{obj} + \text{loss}_{cls}$$

Here, we have the following

- $\text{loss}_x, \text{loss}_y, \text{loss}_w, \text{loss}_h$ are the mean squared error of x, y, w, h
- loss_{obj} is the binary cross-entropy loss of the objectness score
- loss_{cls} is the binary cross-entropy loss of class predictions

In this recipe, you will learn how to implement a combined loss function for the YOLO-v3 algorithm.

How to do it...

We will define the YOLO-v3 loss function in a few steps, as follows:

1. Define the `get_loss_batch` helper function to compute the loss value for a mini-batch:

```
def get_loss_batch(output,targets, params_loss, opt=None):
    ignore_thres=params_loss["ignore_thres"]
    scaled_anchors= params_loss["scaled_anchors"]
    mse_loss= params_loss["mse_loss"]
    bce_loss= params_loss["bce_loss"]
    num_yolos=params_loss["num_yolos"]
    num_anchors= params_loss["num_anchors"]
    obj_scale= params_loss["obj_scale"]
    noobj_scale= params_loss["noobj_scale"]
```

The `get_loss_batch` function will continue by looping over the model outputs:

```
loss=0.0
for yolo_ind in range(num_yolos):
    yolo_out=output[yolo_ind]
    batch_size, num_bbxs, _=yolo_out.shape
    gz_2=num_bbxs/num_anchors
    grid_size=int(np.sqrt(gz_2))
    yolo_out=yolo_out.view(batch_size,num_anchors,grid_size,grid_size,-1)
```

The `get_loss_batch` function will continue by extracting the predicted bounding boxes:

```
pred_boxes=yolo_out[:,:,:,:,4]
x,y,w,h= transform_bbox(pred_boxes,
    scaled_anchors[yolo_ind])
pred_conf=yolo_out[:,:,:,:,4]
pred_cls_prob=yolo_out[:,:,:,:,5:]
```

The `get_loss_batch` function will continue by calling the `get_yolo_targets` helper function:

```
yolo_targets = get_yolo_targets({
    "pred_cls_prob": pred_cls_prob,
    "pred_boxes": pred_boxes,
    "targets": targets,
    "anchors": scaled_anchors[yolo_ind],
```

```
        "ignore_thres":  
        ignore_thres,  
    })
```

Now, we will go back and continue with the `get_loss_batch` helper function:

```
obj_mask=yolo_targets["obj_mask"]  
noobj_mask=yolo_targets["noobj_mask"]  
tx=yolo_targets["tx"]  
ty=yolo_targets["ty"]  
tw=yolo_targets["tw"]  
th=yolo_targets["th"]  
tcls=yolo_targets["tcls"]  
t_conf=yolo_targets["t_conf"]
```

The `get_loss_batch` helper function continues with the following code:

```
loss_x = mse_loss(x[obj_mask], tx[obj_mask])  
loss_y = mse_loss(y[obj_mask], ty[obj_mask])  
loss_w = mse_loss(w[obj_mask], tw[obj_mask])  
loss_h = mse_loss(h[obj_mask], th[obj_mask])
```

The `get_loss_batch` helper function continues with the following code:

```
loss_conf_obj = bce_loss(pred_conf[obj_mask],  
t_conf[obj_mask])  
loss_conf_noobj = bce_loss(pred_conf[noobj_mask],  
t_conf[noobj_mask])  
loss_conf = obj_scale * loss_conf_obj + noobj_scale *  
loss_conf_noobj  
loss_cls = bce_loss(pred_cls_prob[obj_mask],  
tcls[obj_mask])  
loss += loss_x + loss_y + loss_w + loss_h + loss_conf +  
loss_cls
```

The `get_loss_batch` helper function continues with the following code:

```
if opt is not None:  
    opt.zero_grad()  
    loss.backward()  
    opt.step()  
return loss.item()
```

In *step 1*, we called the `transform_bbox` helper function, which is defined as follows:

```
def transform_bbox(bbox, anchors):
    x=bbox[:,:,:,:,0]
    y=bbox[:,:,:,:,1]
    w=bbox[:,:,:,:,2]
    h=bbox[:,:,:,:,3]
    anchor_w = anchors[:, 0].view((1, 3, 1, 1))
    anchor_h = anchors[:, 1].view((1, 3, 1, 1))
    x=x-x.floor()
    y=y-y.floor()
    w= torch.log(w / anchor_w + 1e-16)
    h= torch.log(h / anchor_h + 1e-16)
    return x, y, w, h
```

2. In *step 1*, we called the `get_yolo_targets` helper function, which is defined as follows:

```
def get_yolo_targets(params):
    pred_boxes=params["pred_boxes"]
    pred_cls_prob=params["pred_cls_prob"]
    target=params["targets"]
    anchors=params["anchors"]
    ignore_thres=params["ignore_thres"]

    batch_size = pred_boxes.size(0)
    num_anchors = pred_boxes.size(1)
    grid_size = pred_boxes.size(2)
    num_cls = pred_cls_prob.size(-1)
```

The `get_yolo_targets` helper function will continue by initializing the output tensors:

```
sizeT=batch_size, num_anchors, grid_size, grid_size
obj_mask = torch.zeros(sizeT,device=device,dtype=torch.uint8)
noobj_mask = torch.ones(sizeT,device=device,dtype=torch.uint8)
tx = torch.zeros(sizeT, device=device, dtype=torch.float32)
ty= torch.zeros(sizeT, device=device, dtype=torch.float32)
tw= torch.zeros(sizeT, device=device, dtype=torch.float32)
th= torch.zeros(sizeT, device=device, dtype=torch.float32)
sizeT=batch_size, num_anchors, grid_size, grid_size, num_cls
tcls= torch.zeros(sizeT, device=device, dtype=torch.float32)
```

`get_yolo_targets` continues by scaling and extracting the target bounding boxes:

```
target_bboxes = target[:, 2:] * grid_size
t_xy = target_bboxes[:, :2]
t_wh = target_bboxes[:, 2:]
t_x, t_y = t_xy.t()
t_w, t_h = t_wh.t()

grid_i, grid_j = t_xy.long().t()
```

The `get_yolo_targets` helper function continues by choosing the anchor that has the highest IOU with the targets:

```
iou_with_anchors=[get_iou_WH(anchor, t_wh) for anchor in
anchors]
iou_with_anchors = torch.stack(iou_with_anchors)
best_iou_wa, best_anchor_ind = iou_with_anchors.max(0)
```

The `get_yolo_targets` helper function continues by setting the object mask tensors:

```
batch_inds, target_labels = target[:, :2].long().t()
obj_mask[batch_inds, best_anchor_ind, grid_j, grid_i] = 1
noobj_mask[batch_inds, best_anchor_ind, grid_j, grid_i] = 0

for ind, iou_wa in enumerate(iou_with_anchors.t()):
    noobj_mask[batch_inds[ind], iou_wa > ignore_thres,
    grid_j[ind], grid_i[ind]] = 0
```

The `get_yolo_targets` helper function continues by setting x and y:

```
tx[batch_inds, best_anchor_ind, grid_j, grid_i] = t_x - t_x.floor()
ty[batch_inds, best_anchor_ind, grid_j, grid_i] = t_y - t_y.floor()
```

The `get_yolo_targets` helper function continues by setting w and h:

```
anchor_w=anchors[best_anchor_ind][:, 0]
tw[batch_inds, best_anchor_ind, grid_j, grid_i] = torch.log(t_w /
anchor_w + 1e-16)
anchor_h=anchors[best_anchor_ind][:, 1]
th[batch_inds, best_anchor_ind, grid_j, grid_i] = torch.log(t_h /
anchor_h + 1e-16)
```

The `get_yolo_targets` helper function continues by setting target classes:

```
tcls[batch_inds, best_anchor_ind, grid_j, grid_i, target_labels] = 1
```

Finally, the `get_yolo_targets` helper function returns the output as a Python dictionary:

```
output={  
    "obj_mask" : obj_mask,  
    "noobj_mask" : noobj_mask,  
    "tx": tx,  
    "ty": ty,  
    "tw": tw,  
    "th": th,  
    "tcls": tcls,  
    "t_conf": obj_mask.float(),  
}  
return output
```

3. Next, we define the `get_iou_WH` helper function:

```
def get_iou_WH(wh1, wh2):  
    wh2 = wh2.t()  
    w1, h1 = wh1[0], wh1[1]  
    w2, h2 = wh2[0], wh2[1]  
    inter_area = torch.min(w1, w2) * torch.min(h1, h2)  
    union_area = (w1 * h1 + 1e-16) + w2 * h2 - inter_area  
    return inter_area / union_area
```

In the next section, we will explain how each step works in detail.

How it works...

In *step 1*, we defined the `get_loss_batch` helper function. The function inputs were as follows:

- `output`: A list of three tensors corresponding to the YOLO-v3 outputs.
- `targets`: The ground truth, a tensor of shape $n \times 6$, where n is the total number of bounding boxes in the batch.
- `params_loss`: A Python dict, which contains the loss parameters.
- `opt`: An object of the optimizer class. The default value is `None`.

As you can see, we broke down the helper function into multiple snippets for better code readability. Initially, we extracted the parameters from the `params_loss` dictionary. Then, we created a loop of `num_yolos=3` iterations. In each iteration, we extracted the YOLO output (`yolo_out`), the number of bounding boxes (`num_bboxes`), and the grid size (`grid_size`).



As a reminder, there were three YOLO outputs with 507, 2028, 8112 bounding boxes in each output, respectively. Also, the grid sizes were 13, 26, 52, respectively.

Then, we reshaped `yolo_out` to `(batch_size, 3, grid_size, grid_size, 85)` in each iteration. Here, we had $3 \times 13 \times 13 = 507$, $3 \times 26 \times 26 = 2028$, and $3 \times 52 \times 52 = 8112$. From the reshaped tensor, we got the predicted bounding boxes (`pred_boxes`), objectness score (`pred_conf`), and the class probabilities (`pred_cls_prob`). We also transformed the predicted bounding boxes using the `transform_bbox` helper function. We defined the `transform_bbox` helper function in *step 2*.

Next, we passed the `params_target` dictionary to the `get_yolo_targets` helper function to obtain `yolo_targets`. We defined the `get_yolo_targets` function in *step 3*. The output of the `get_yolo_targets` function was a dictionary that contained the variables we needed to compute the loss value.

Next, we calculated the mean-squared error (`mse_loss`) between the predicted and the target coordinates of the bounding boxes (`loss_x`, `loss_y`, `loss_w`, `loss_h`). Then, we calculated the binary cross-entropy loss (`bce_loss`) between the predicted and target objectness scores. We also calculated the `bce_loss` for the predicted class probabilities and target labels. Finally, all the calculated loss values were summed together. This was repeated three times and the loss value of each iteration was added to that of the previous iteration.

If the optimization object, `opt`, has a value, we compute the gradients and perform the optimization step; otherwise, we return the total loss.



The optimizer will be used during training to update the model parameters.

In *step 2*, we defined the `transform_bbox` helper function. The inputs to the helper function were as follows:

- `bbox`: A tensor of shape `(batch_size, 3, grid_size, grid_size, 4)` that contains the predicted bounding boxes.
- `anchors`: A tensor of shape `(3, 2)` that contains the scaled widths and heights of the three anchors for each YOLO output. For example, the width and height of the three anchors in the first YOLO output, are `[[116, 90], [156, 198], [373, 326]]` and their scaled values are `[[3.6250, 2.8125], [4.8750, 6.1875], [11.6562, 10.1875]]`. Here, the scale factor is 32.

This function takes the predicted bounding boxes and transforms them so that they're compatible with the target values. This transformation is the reverse of the `transform_outputs` function from the `YOLOLayer` class that we defined in the *Creating the YOLOv3 model* recipe.

In the function, we extracted the `x`, `y`, `w`, `h` tensors from `bbox`. Then, we sliced the two columns of the `anchors` tensor and reshaped them into tensors of shape `(1, 3, 1, 1)` using the `.view()` method. Next, we transformed the values and returned a list of four tensors corresponding to `x`, `y`, `w`, `h`.

In *step 3*, we defined the `get_yolo_targets` helper function. The input to the function was a Python dictionary with the following keys:

- `pred_boxes`: A tensor of shape `(batch_size, 3, grid_size, grid_size, 4)` that contains the predicted bounding boxes.
- `pred_cls_prob`: A tensor of shape `(batch_size, 3, grid_size, grid_size, 80)` that contains the predicted class probabilities.
- `targets`: A tensor of shape `(n, 6)`, where `n` is the number of bounding boxes in a batch, containing the ground truth bounding boxes and labels.
- `anchors`: A tensor of shape `(2, 3)` that contains the scaled width and height of the three anchors.
- `ignore_thres`: A scalar float value set to `0.5`, which is used as the threshold value.

In the function, we extracted the keys and initialized the output tensors.

The `obj_mask` and `noobj_mask` tensors will be used later to filter the coordinates of the bounding boxes when calculating the loss function. Then, we sliced the target bounding boxes from the `target` tensor and scaled by the grid size. This was due to the fact that the original target bounding boxes were normalized. The new tensor, `target_bboxes`, of shape `(n, 4)` contains the scaled coordinates of the bounding boxes `[x, y, w, h]`. Next, we extracted `x`, `y`, `w`, `h` into four individual tensors of `n` items.



The `.t()` method is used to transpose a tensor. For example, if `A` is a `2x3` tensor, `A.t()` will be a `3x2` tensor.

Next, we calculated the **intersection over union (IoU)** of a target and the three anchors using the `get_iou_WH` helper function. Then, we found the anchor that has the highest IOU with a target. The `get_iou_WH` helper function was defined in *step 4*.



The `.long()` method is used to convert float values into integer values.

Next, we extracted the batch index and object labels from the `target` tensor. The `batch_inds` variable holds the indices of images in the batch, which range from `0` to `batch_size-1`. The `target_labels` variable holds the object classes, which range from `0` to `79` since the total number of object classes is `80`. Next, we updated the `obj_mask`, `noobj_mask`, `tx`, `ty`, `tw`, `th`, and `tcls` tensors.



The `tx` and `ty` tensors were updated so that they could hold values in the range of `(0, 1)`.

In *step 4*, we defined the `get_iou_WH` helper function. The inputs to the helper function are as follows:

- `wh1`: A tensor of shape `(1, 2)`, which contains the width and height of an anchor
- `wh2`: A tensor of shape `(n, 2)`, which contains the width and height of the target bounding boxes

Here, we transposed `wh2` into a tensor of shape `(2, n)`. Then, we calculated the intersection and union of the targets with the anchor and returned IOU values.

Training the model

So far, we've created dataloaders and defined the model and the loss function. In this recipe, we will develop the code to train the model. The training process will follow the standard **stochastic gradient descent (SGD)** process. We will train the model on the training data and evaluate it on the validation data. Due to the large size of the COCO dataset and our deep model of the YOLO-v3 architecture, training will be very slow, even when using a GPU. You may need to train the model for up to a week to get good performance.

How to do it...

In this recipe, we will define a few helper functions, set the necessary hyperparameters, and train the model. Let's get started:

1. Define the `loss_epoch` helper function to compute the loss function for each epoch:

```
def loss_epoch(model, params_loss, dataset_dl, sanity_check=False, opt=None):
    running_loss=0.0
    len_data=len(dataset_dl.dataset)
    running_metrics= {}
    for xb, yb, _ in dataset_dl:
        yb=yb.to(device)
        _,output=model(xb.to(device))
        loss_b=get_loss_batch(output,yb, params_loss,opt)
        running_loss+=loss_b
        if sanity_check is True:
            break
    loss=running_loss/float(len_data)
    return loss
```

2. Define the `train_val` helper function to train the model:

```
import copy
def train_val(model, params):
    num_epochs=params["num_epochs"]
    params_loss=params["params_loss"]
```

```

opt=params["optimizer"]
train_dl=params["train_dl"]
val_dl=params["val_dl"]
sanity_check=params["sanity_check"]
lr_scheduler=params["lr_scheduler"]
path2weights=params["path2weights"]

```

The `train_val` helper function continues with the following code:

```

loss_history={
    "train": [],
    "val": [],
}
best_model_wts = copy.deepcopy(model.state_dict())
best_loss=float('inf')

```

The `train_val` helper function continues with the following code:

```

for epoch in range(num_epochs):
    current_lr=get_lr(opt)
    print('Epoch {} / {}, current lr={}'.format(epoch, num_epochs - 1, current_lr))
    model.train()
    train_loss=loss_epoch(model,params_loss,train_dl,sanity_check,opt)
    loss_history["train"].append(train_loss)
    print("train loss: %.6f" %(train_loss))

```

The `train_val` helper function continues with the following code:

```

model.eval()
with torch.no_grad():
    val_loss=loss_epoch(model,params_loss,val_dl,sanity_check)
    loss_history["val"].append(val_loss)
    print("val loss: %.6f" %(val_loss))

```

The `train_val` helper function continues with the following code:

```

if val_loss < best_loss:
    best_loss = val_loss
    best_model_wts = copy.deepcopy(model.state_dict())
    torch.save(model.state_dict(), path2weights)
    print("Copied best model weights!")

```

The `train_val` helper function continues with the following code:

```

lr_scheduler.step(val_loss)
if current_lr != get_lr(opt):
    print("Loading best model weights!")
    model.load_state_dict(best_model_wts)

```

```

        print("*****")
model.load_state_dict(best_model_wts)
return model, loss_history

```

Here, the `get_lr` helper function is defined as follows:

```

def get_lr(opt):
    for param_group in opt.param_groups:
        return param_group['lr']

```

3. Before calling the `train_val` function, let's define the optimizer:

```

from torch import optim
from torch.optim.lr_scheduler import ReduceLROnPlateau

opt = optim.Adam(model.parameters(), lr=1e-3)
lr_scheduler = ReduceLROnPlateau(opt, mode='min', factor=0.5,
                                patience=20, verbose=1)

path2models= "./models/"
if not os.path.exists(path2models):
    os.mkdir(path2models)
scaled_anchors=[model.module_list[82][0].scaled_anchors,
               model.module_list[94][0].scaled_anchors,
               model.module_list[106][0].scaled_anchors]

```

Then, we will set the loss parameters:

```

mse_loss = nn.MSELoss(reduction="sum")
bce_loss = nn.BCELoss(reduction="sum")
params_loss={
    "scaled_anchors" : scaled_anchors,
    "ignore_thres": 0.5,
    "mse_loss": mse_loss,
    "bce_loss": bce_loss,
    "num_yolos": 3,
    "num_anchors": 3,
    "obj_scale": 1,
    "noobj_scale": 100,
}

```

Next, we will set the training parameters and call the `train_val` function:

```

params_train={
    "num_epochs": 20,
    "optimizer": opt,
    "params_loss": params_loss,
    "train_dl": train_dl,
}

```

```
        "val_dl": val_dl,
        "sanity_check": False,
        "lr_scheduler": lr_scheduler,
        "path2weights": path2models+"weights.pt",
    }
model, loss_hist=train_val(model,params_train)
```

Depending on your computer's capability, you may get an out-of-memory error while running the preceding snippet.



If you get an out of memory error on your computer, try to reduce the batch size and restart the notebook.

Training will start and you should see its progress, as shown in the following snippet:

```
Epoch 0/99, current lr=0.001
train loss: 2841.816801
val loss: 689.142467
Copied best model weights!
-----
Epoch 1/99, current lr=0.001
train loss: 699.876479
val loss: 659.404275
Copied best model weights!
-----
```

In the next section, we explain how each step works in detail.

How it works...

In *step 1*, we defined the `loss_epoch` helper function to compute the loss value in an epoch.

The inputs to this function are as follows:

- `model`: An object of the model class
- `param_loss`: A Python dictionary that holds the parameters of the loss function
- `dataset_dl`: An object of the dataloader class
- `sanity_check`: A flag with a default value of `False` to break the loop after one batch
- `opt`: An object of the optimization class with a default value of `None`

In this function, we extracted batches of data and target values using the dataloader. Then, we fed the data to the model and obtained the second output.



The model returns two outputs. The first output is a tensor of shape `(batch_size, 10647, 85)`, which is the concatenation of three YOLO outputs. The second output is a list of three tensors that correspond to the three YOLO outputs.

Next, we computed the loss value for each batch using the `get_loss_batch` function, which was defined in the *Defining the loss function* recipe. If the `sanity_check` flag is `True`, we break the loop; otherwise, we continue the loop and return the average loss value.

In *step 2*, we defined the `train_val` helper function. The inputs to the function are as follows:

- `model`: An object of the model class
- `params`: A Python dictionary that contains the training parameters

In this function, we extracted the `params` keys and initialized a few variables. Then, the training loop started. In each iteration of the loop, we trained the model for one epoch on the training data and computed the training loss for each epoch using the `loss_epoch` helper function. Next, we evaluated the model on the validation data and computed the validation loss. Notice that, for the validation data, we did not pass the `opt` argument to the `loss_epoch` function. If the validation loss improved in an iteration, we stored a copy of the model weights. We also used the learning scheduler to monitor the loss and reduce the learning rate in the case of plateaus. In the end, the trained model and the loss history were returned.

We also defined the `get_lr` helper function, which is a simple function that returns the learning rate in each iteration of the training loop.

In *step 3*, we got ready to train the model. First, we defined the optimizer and learning rate schedule. Then, we defined the loss and training parameters. Next, we called the `train_val` function. Due to the large COCO dataset and the YOLO-v3 model, the training will be very slow, even using a GPU. You'll need to train the model for up to a week. It is better to set the `sanity_check` flag to `True` first to quickly run through the training loop in a short amount of time and fix any possible errors. Then, you can set the flag back to `False` and train the model using the complete dataset.

Deploying the model

Now that training is complete, it is time to deploy the model. To deploy the model, we need to define the model class, as described in the *Creating the YOLOv3 model* recipe. Then, we need to load the trained weights into the model and deploy it on the validation dataset. Let's learn how to do this.

In this recipe, you will learn how to load weights into the model, deploy it on a sample image, and display the results.

How to do it...

In this recipe, we will load weights and deploy the model on a sample image. Let's get started:

1. Load trained weights into the model:

```
path2weights = "./models/weights.pt"
model.load_state_dict(torch.load(path2weights))
```

2. Get a sample image from the validation dataset:

```
img, tg, _ = coco_val[4]
print(img.shape)
print(tg.shape)
```

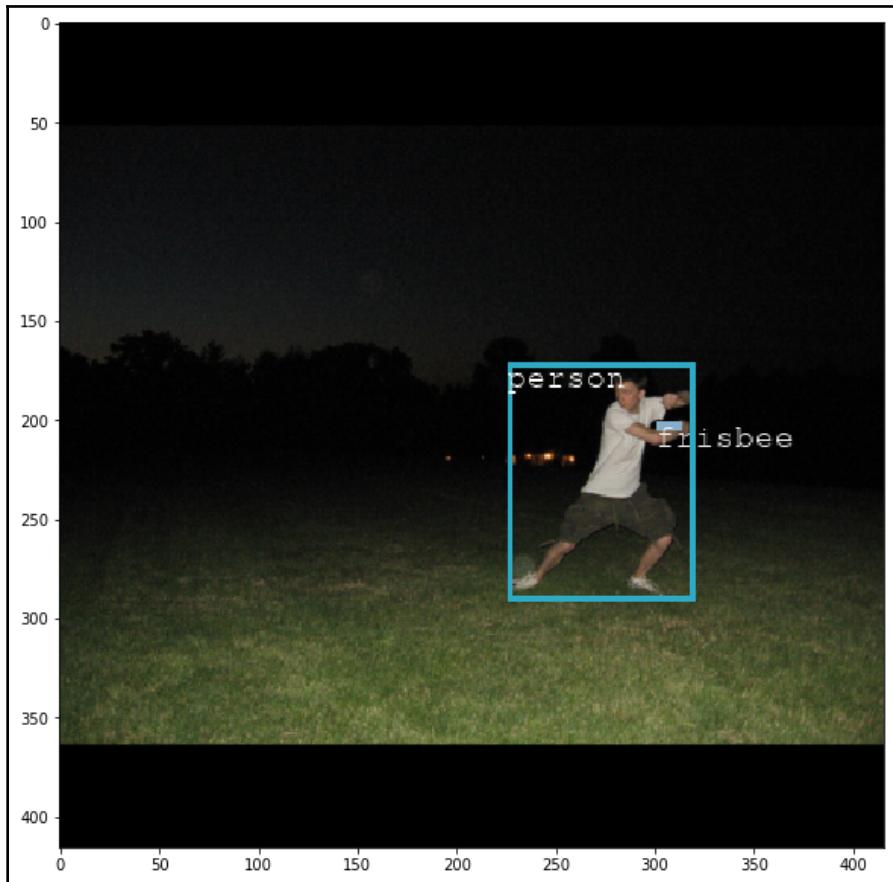
The preceding snippet will print the following output:

```
torch.Size([3, 416, 416])
torch.Size([2, 6])
```

3. Let's display the image and its bounding boxes:

```
show_img_bbox(img, tg)
```

The preceding snippet will return the following screenshot:



As we can see, the image has a person and a frisbee.

4. Now, let's feed the image to the model to get the output:

```
model.eval()
with torch.no_grad():
    out,_=model(img.unsqueeze(0).to(device))
print(out.shape)
```

The preceding snippet will print the following output:

```
torch.Size([1, 10647, 85])
```

5. Pass the model's output to the NonMaxSuppression function:

```
img_size=416  
out_nms=NonMaxSuppression(out.cpu())  
print (out_nms[0].shape)
```

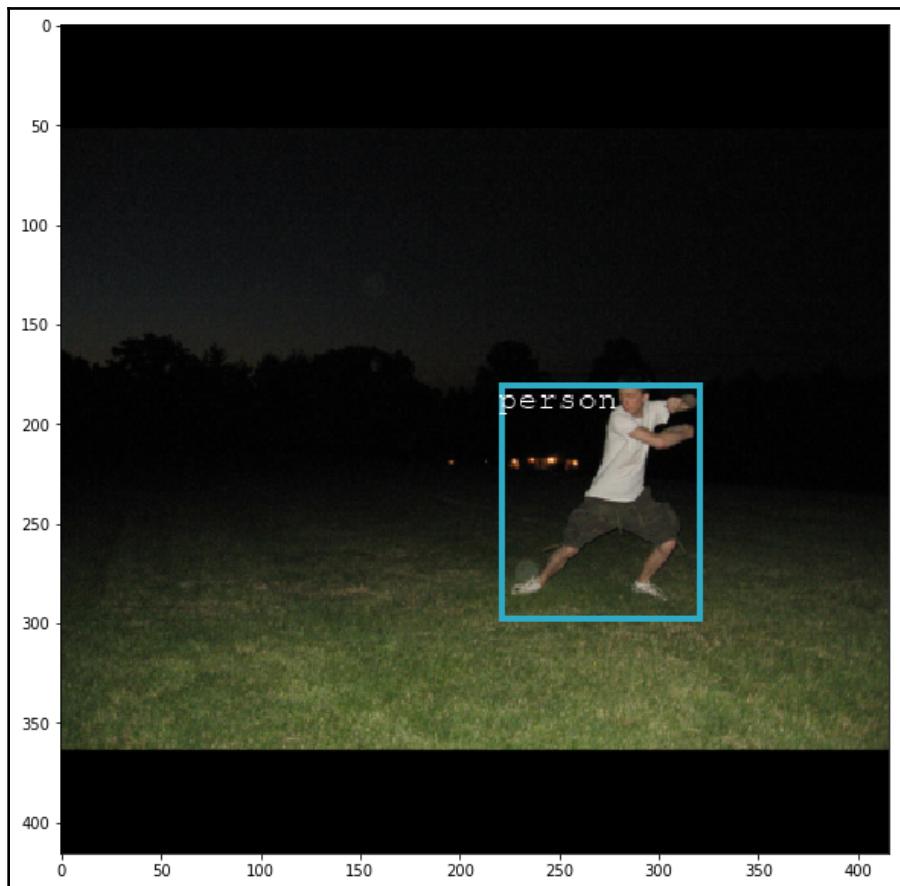
The preceding snippet will print the following output:

```
torch.Size([1, 6])
```

6. Let's show the image and predicted bounding box:

```
show_img_bbox(img,out_nms[0])
```

The preceding snippet will display the following screenshot:



As we can see, the person in the image was identified and located using a bounding box.

7. The NonMaxSuppression function is defined as follows:

```
def NonMaxSuppression(bbox_pred, obj_threshold=0.5, nms_thres=0.5):
    bbox_pred[..., :4] = xywh2xyxy(bbox_pred[..., :4])
    output = [None] * len(bbox_pred)
    for ind, bb_pr in enumerate(bbox_pred):
        bb_pr = bb_pr[bb_pr[:, 4] >= obj_threshold]
        if not bb_pr.size(0):
            continue
```

The NonMaxSuppression function continues with the following code:

```
score = bb_pr[:, 4] * bb_pr[:, 5:].max(1)[0]
bb_pr = bb_pr[(-score).argsort()]
cls_probs, cls_preds = bb_pr[:, 5:].max(1, keepdim=True)
detections = torch.cat((bb_pr[:, :5],
                        cls_probs.float(),
                        cls_preds.float()), 1)
```

The NonMaxSuppression function continues with the following code:

```
bbox_nms = []
while detections.size(0):
    high_iou_inds = bbox_iou(detections[0,
                                         :4].unsqueeze(0),
                                         detections[:, :4]) > nms_thres
    cls_match_inds = detections[0, -1] == detections[:, -1]
    supp_inds = high_iou_inds & cls_match_inds
    ww = detections[supp_inds, 4]
    detections[0, :4] = (ww * detections[supp_inds,
                                         :4]).sum(0) / ww.sum()
    bbox_nms += [detections[0]]
    detections = detections[~supp_inds]
```

The NonMaxSuppression function ends by returning the output:

```
if bbox_nms:
    output[ind] = torch.stack(bbox_nms)
    output[ind] = xyxyh2xywh(output[ind])
return output
```

8. `xywh2xyxy` is defined as follows:

```
def xywh2xyxy(xywh):
    xyxy = xywh.new(xywh.shape)
    xyxy[..., 0] = xywh[..., 0] - xywh[..., 2] / 2.0
    xyxy[..., 1] = xywh[..., 1] - xywh[..., 3] / 2.0
    xyxy[..., 2] = xywh[..., 0] + xywh[..., 2] / 2.0
    xyxy[..., 3] = xywh[..., 1] + xywh[..., 3] / 2.0
    return xyxy
```

9. `xyxyh2xywh` is defined as follows:

```
def xyxyh2xywh(xyxy, image_size=416):
    xywh = torch.zeros(xyxy.shape[0], 6)
    xywh[:, 2] = (xyxy[:, 0] + xyxy[:, 2]) / 2./img_size
    xywh[:, 3] = (xyxy[:, 1] + xyxy[:, 3]) / 2./img_size
    xywh[:, 5] = (xyxy[:, 2] - xyxy[:, 0])/img_size
    xywh[:, 4] = (xyxy[:, 3] - xyxy[:, 1])/img_size
    xywh[:, 1]= xyxy[:, 6]
    return xywh
```

10. `bbox_iou` is defined as follows:

```
def bbox_iou(box1, box2):
    b1_x1, b1_y1, b1_x2, b1_y2 = box1[:, 0], box1[:, 1], box1[:, 2], box1[:, 3]
    b2_x1, b2_y1, b2_x2, b2_y2 = box2[:, 0], box2[:, 1], box2[:, 2], box2[:, 3]
    inter_rect_x1 = torch.max(b1_x1, b2_x1)
    inter_rect_y1 = torch.max(b1_y1, b2_y1)
    inter_rect_x2 = torch.min(b1_x2, b2_x2)
    inter_rect_y2 = torch.min(b1_y2, b2_y2)
    inter_area = torch.clamp(inter_rect_x2 - inter_rect_x1 + 1,
    min=0) \
                *torch.clamp(inter_rect_y2 - inter_rect_y1 + 1,
    min=0)
    b1_area = (b1_x2 - b1_x1 + 1.0) * (b1_y2 - b1_y1 + 1.0)
    b2_area = (b2_x2 - b2_x1 + 1.0) * (b2_y2 - b2_y1 + 1.0)
    iou = inter_area / (b1_area + b2_area - inter_area + 1e-16)
    return iou
```

In the next section, we will explain how each step works in detail.

How it works...

In *step 1*, we loaded the trained weights into the YOLOv3 model. Make sure that the YOLOv3 model is defined as described in the *Creating the YOLOv3 model* recipe before loading the weights.

In *step 2*, we selected a sample image from the validation dataset and printed the image and its target shape. As expected, the image was a tensor of shape (3, 416, 416). The target shape was (2, 6), representing two bounding boxes.



A row in the target tensor contains [img_ind, obj_id, x, y, w, h], where img_ind is the index of the image in the batch and obj_id is the object label or class.

In *step 3*, we displayed the image and corresponding bounding boxes using the show_img_bbox helper function that was defined in the *Creating a custom COCO dataset* recipe.

In *step 4*, we set the model in evaluation mode and fed the image to the model, got the first output, and printed the output shape. As we saw, there were 10647 detected bounding boxes. However, not all of the detected bounding boxes are valid and they should be filtered. Filtering was performed in *step 5* using the NonMaxSuppression function. The function returned only one valid bounding box. We displayed the predicted bounding box in *step 6*. It seems the model was only able to detect the person in the image.

In *step 7*, we defined the NonMaxSuppression function. In a nutshell, the non-max suppression algorithm selects a bounding box with the highest probability and removes any bounding boxes that have a large overlap with the selected bounding box.

The inputs to the function are as follows:

- bbox_pred: A tensor of shape (batch_size, 10647, 85), coming from the model output
- obj_threshold: A scalar value, the threshold at which we compare the predicted objectness score for each bounding box
- nms_threshold: A scalar value, the IOU threshold at which we suppress the bounding boxes

In the function, we converted the predicted bounding boxes in `[xc, yc, w, h]` format into `[x1, y1, x2, y2]` format. Then, we created a loop to read the bounding boxes for each image. Thus, `bb_pr` holds the detected bounding boxes per image and has a shape of `(10647, 85)`. Next, we compared the predicted objectness score at index `4` of `bb_pr` with `obj_threshold` to filter out bounding boxes with a low probability. Next, we multiplied the objectness probability with the maximum class probability and called it `score`. Then, we sorted (descending) the remaining bounding boxes based on `score`. Next, we calculated the IOU between the highest score bounding box at index `0` and the other bounding boxes and found the indices (`high_iou_inds`) that were larger than `nms_threshold`. We also found the indices of the bounding boxes that have the same class prediction (`cls_match_inds`). The intersection of `high_iou_inds` and `cls_match_inds` holds the indices of the bounding boxes with the same class prediction and high overlap, which were suppressed. Finally, we converted the filtered bounding boxes from `[x1, y1, x2, y2]` into `[xc, yc, w, h]` format and returned the output.

In *step 8*, we defined the `xywh2xyxy` helper function. The function's input is `xywh`, a tensor of shape `(n, 4)` that holds `n` bounding boxes in `[xc, yc, w, h]` format.

The function returns a tensor of shape `(n, 4)`, which holds `n` bounding boxes in `[x1, y1, x2, y2]` format.

In *step 9*, we defined the `xyxy2xywh` helper function. The function input is `xyxy`, a tensor of shape `(n, 4)` that holds `n` bounding boxes in `[x1, y1, x2, y2]` format.

The function returns a tensor of shape `(n, 4)`, which holds `n` bounding boxes in `[xc, yc, w, h]` format.

In *step 10*, we defined the `bbox_iou` helper function. The inputs of this function are as follows:

- `box1`: A tensor of shape `(1, 4)`, which contains one bounding box in `[x1, y1, x2, y2]` format
- `box2`: A tensor of shape `(n, 4)`, which contains `n` bounding boxes in `[x1, y1, x2, y2]` format

In this function, we calculated the area of the intersection. Then, we computed the area of the union and returned the IOU.

See also

While writing this chapter, we were inspired by the following implementations:

- <https://pjreddie.com/darknet/yolo/>
- <https://blog.paperspace.com/tag/series-yolo/>
- <https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b>
- <https://github.com/eriklindernoren/PyTorch-YOLOv3>
- <https://github.com/ayooshkathuria/pytorch-yolo-v3>

Check these links out for more insights into the YOLO model.

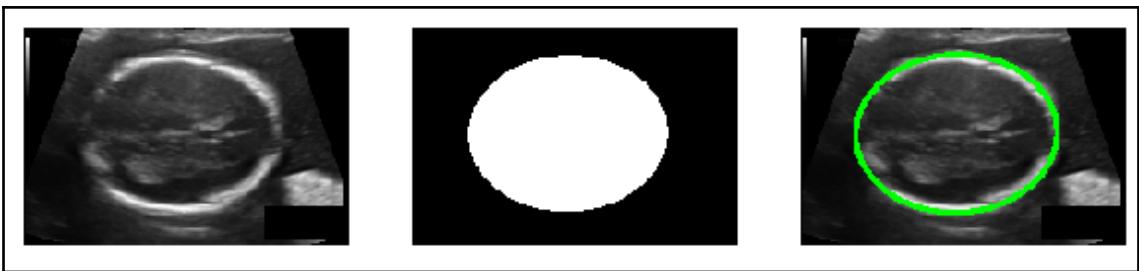
6

Single-Object Segmentation

Object segmentation is the process of finding the boundaries of target objects in images. There are many applications for segmenting objects in images. As an example, by outlining anatomical objects in medical images, clinical experts can learn useful information about patients' conditions.

Depending on the number of objects in images, we can deal with single-object or multi-object segmentation tasks. This chapter will focus on developing a deep-learning model using PyTorch to perform single-object segmentation. In single-object segmentation, we are interested in automatically outlining the boundary of one target object in an image.

The object boundary is usually defined by a binary mask. From the binary mask, we can overlay a contour on the image to outline the object boundary. As an example, the following screenshot depicts an ultrasound image of a fetus, a binary mask corresponding to the fetal head, and the segmentation of the fetal head overlaid on the ultrasound image:



Other examples include the segmentation of heart chambers in MRI or CT images in order to calculate multiple clinical indices including the ejection fraction.

The goal of automatic single-object segmentation is to predict a binary mask given in an image. In this chapter, we will learn how to create an algorithm to automatically segment a fetal head in ultrasound images. This is an important task in medical imaging in order to measure the fetal head circumference.

In this chapter, we will cover the following recipes:

- Creating custom datasets
- Defining the model
- Defining the loss function and optimizer
- Training the model
- Deploying the model

Creating custom datasets

We will use the data from the **Automated measurement of fetal head circumference** competition on the **Grand Challenge** website. During pregnancy, ultrasound imaging is used to measure the fetal head circumference. The measurement can be used to monitor the growth of the fetus. The dataset contains the two-dimensional (2D) ultrasound images of the standard plane.

In this recipe, we will use the `Dataset` class from the `torch.utils.data` package to create custom datasets for loading and processing data.

Getting ready

To download the dataset, visit <https://zenodo.org/record/1322001#.XcX1jk9KhhE> and go through the following steps:

1. Download the `training_set.zip` and `test_set.zip` files.
2. Move the ZIP files to a folder named `data` in the same location as your code and extract them as `training_set` and `test_set`, respectively. The `training_set` folder should contain 1,998 `.png` files, including 999 images and 999 annotations. In addition, the `test_set` folder should contain 335 `.png` images. There are no annotation files in the `test_set` folder.

We will use the images and annotations in the `training_set` folder for training the model.

Then, we will deploy the model on the images in the `test_set` folder.

How to do it...

In this recipe, we will explore the data, define augmentation functions, and create the custom datasets.

Data exploration

We will first explore the data and display a few sample images and masks:

1. Let's inspect the `training_set` folder by counting the number of images and annotations:

```
import os
path2train="../data/training_set/"

imgsList=[pp for pp in os.listdir(path2train) if "Annotation" not
in pp]
anntsList=[pp for pp in os.listdir(path2train) if "Annotation" in
pp]
print("number of images:", len(imgsList))
print("number of annotations:", len(anntsList))
```

The preceding code snippet will print the following output:

```
number of images: 999
number of annotations: 999
```

2. Next, we will display a few sample images and masks:

Select random images from the list using the following code:

```
import numpy as np
np.random.seed(2019)
rndImgs=np.random.choice(imgsList,4)
rndImgs
```

The preceding code snippet will print the following output:

```
array(['425_HC.png', '804_HC.png', '603_HC.png', '082_HC.png'],
      dtype='<U11')
```

Define a helper function to show an image and its annotation:

```
import matplotlib.pyplot as plt
from PIL import Image
from scipy import ndimage as ndi
from skimage.segmentation import mark_boundaries
```

```
def show_img_mask(img, mask):
    img_mask=mark_boundaries(np.array(img),
                            np.array(mask),
                            outline_color=(0,1,0),
                            color=(0,1,0))
    plt.imshow(img_mask)
```

Call the helper function to display the images and masks:

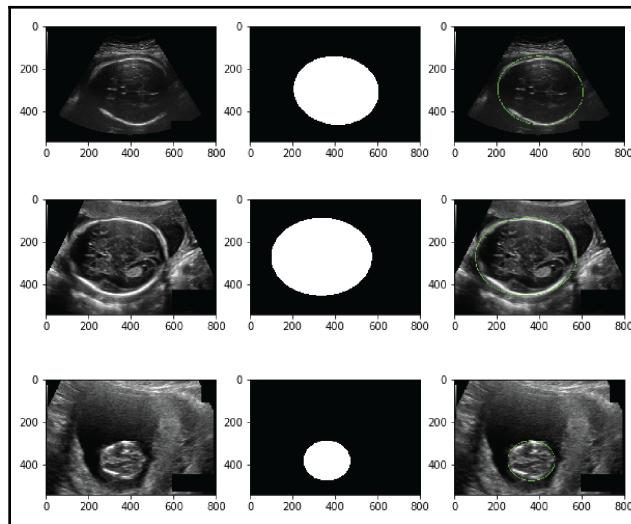
```
for fn in rndImgs:
    path2img = os.path.join(path2train, fn)
    path2annt= path2img.replace(".png", "_Annotation.png")
    img = Image.open(path2img)
    annt_edges = Image.open(path2annt)
    mask = ndi.binary_fill_holes(annt_edges)

    plt.figure()
    plt.subplot(1, 3, 1)
    plt.imshow(img, cmap="gray")

    plt.subplot(1, 3, 2)
    plt.imshow(mask, cmap="gray")

    plt.subplot(1, 3, 3)
    show_img_mask(img, mask)
```

The preceding code snippet will produce different outputs that are shown in the following image:



We have explored our data. Let's now look at a way to augment the data.

Data augmentation

We will use the `albumentations` package to augment the data for segmentation tasks:

1. Install the `albumentations` package in your `conda` environment:

```
$ conda install -c conda-forge imgaug
$ conda install albumentations -c albumentations
```

2. Import the augmentation functions:

```
from albumentations import (
    HorizontalFlip,
    VerticalFlip,
    Compose,
    Resize
)
```

3. Define `transform_train` for the training dataset:

```
h,w=128,192
transform_train = Compose([
    Resize(h,w),
    HorizontalFlip(p=0.5),
    VerticalFlip(p=0.5),
])
```

4. Define `transform_val` for the validation dataset:

```
transform_val = Resize(h,w)
```

We will pass the transformer function to the custom dataset class as defined in the next subsection.

Creating the datasets

We will now create the training and validation datasets:

1. Import the packages and define the dataset class:

```
from torch.utils.data import Dataset
from PIL import Image
from torchvision.transforms.functional import to_tensor,
to_pil_image
```

Define the fetal_dataset class:

```
class fetal_dataset(Dataset):
    def __init__(self, path2data, transform=None):
        imgsList=[pp for pp in os.listdir(path2data) if
"Annotation" not in pp]
        anntsList=[pp for pp in os.listdir(path2train) if
"Annotation" in pp]

        self.path2imgs = [os.path.join(path2data, fn) for fn in
imgsList]
        self.path2annts= [p2i.replace(".png", "_Annotation.png")
for p2i in self.path2imgs]
        self.transform = transform
    def __len__(self):
        return len(self.path2imgs)
```

The dataset class will continue as follows:

```
def __getitem__(self, idx):
    path2img = self.path2imgs[idx]
    image = Image.open(path2img)

    path2annt = self.path2annts[idx]
    annt_edges = Image.open(path2annt)
    mask = ndi.binary_fill_holes(annt_edges)
    image= np.array(image)
    mask=mask.astype("uint8")
```

The dataset class will continue as follows:

```
if self.transform:
    augmented = self.transform(image=image, mask=mask)
    image = augmented['image']
    mask = augmented['mask']

    image= to_tensor(image)
```

```
mask=255*to_tensor(mask)
return image, mask
```

2. Define two objects of the fetal_dataset class:

```
fetal_ds1=fetal_dataset(path2train, transform=transform_train)
fetal_ds2=fetal_dataset(path2train, transform=transform_val)
print(len(fetal_ds1))
print(len(fetal_ds2))
```

The preceding code snippet will print the following output:

```
999
999
```

3. Fetch a sample image and mask from the feta_ds1 and display it:

```
img,mask=fetal_ds1[0]
print(img.shape, img.type(), torch.max(img))
print(mask.shape, mask.type(), torch.max(mask))
```

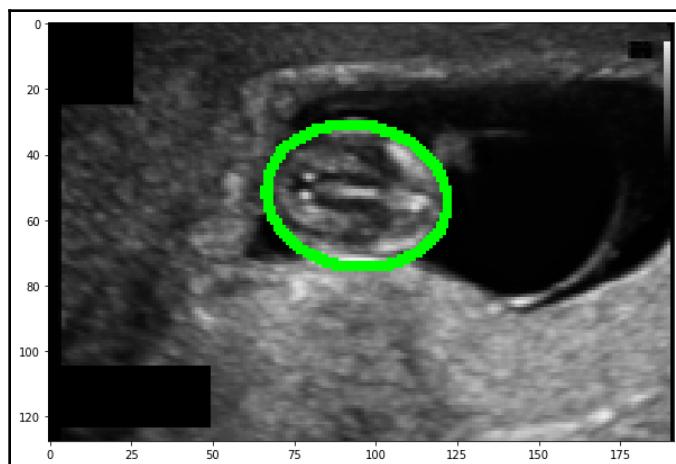
The preceding code snippet will print the following output:

```
torch.Size([1, 128, 192]) torch.FloatTensor tensor(0.9373)
torch.Size([1, 128, 192]) torch.FloatTensor tensor(1.)
```

Display the image and mask:

```
show_img_mask(img, mask)
```

The preceding code snippet will display the following image:



4. Split the data into two groups:

```
from sklearn.model_selection import ShuffleSplit

sss = ShuffleSplit(n_splits=1, test_size=0.2, random_state=0)

indices=range(len(fetal_ds1))

for train_index, val_index in sss.split(indices):
    print(len(train_index))
    print("-"*10)
    print(len(val_index))
```

The preceding code snippet will print the following output:

```
799
-----
200
```

5. Create `train_ds` and `val_ds`:

```
from torch.utils.data import Subset

train_ds=Subset(fetal_ds1,train_index)
print(len(train_ds))

val_ds=Subset(fetal_ds2,val_index)
print(len(val_ds))
```

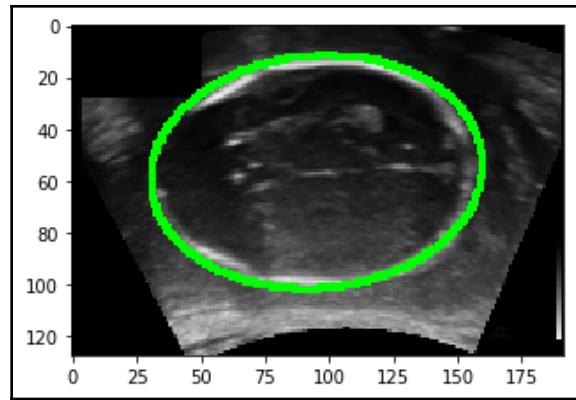
The preceding code snippet will print the following output:

```
799
200
```

6. Show a sample image from `train_ds`:

```
plt.figure(figsize=(5,5))
for img,mask in train_ds:
    show_img_mask(img,mask)
    break
```

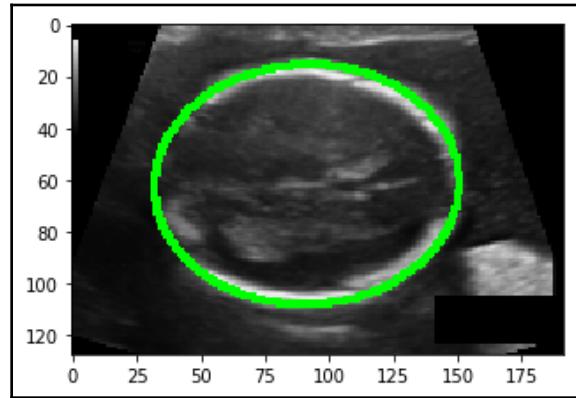
The preceding code snippet will display the following screenshot:



Show a sample image and mask from val_ds:

```
plt.figure(figsize=(5, 5))
for img,mask in val_ds:
    show_img_mask(img,mask)
    break
```

The preceding code snippet will display the following screenshot:



7. Define the data loaders:

```
from torch.utils.data import DataLoader

train_dl = DataLoader(train_ds, batch_size=8, shuffle=True)
val_dl = DataLoader(val_ds, batch_size=16, shuffle=False)
```

8. Get a sample batch from `train_dl`:

```
for img_b, mask_b in train_dl:  
    print(img_b.shape, img_b.dtype)  
    print(mask_b.shape, mask_b.dtype)  
    break
```

The preceding code snippet will print the following output:

```
torch.Size([8, 1, 128, 192]) torch.uint8  
torch.Size([8, 1, 128, 192]) torch.uint8
```

Get a sample batch from `val_dl`:

```
for img_b, mask_b in val_dl:  
    print(img_b.shape, img_b.dtype)  
    print(mask_b.shape, mask_b.dtype)  
    break
```

The preceding code snippet will print out the following output:

```
torch.Size([16, 1, 128, 192]) torch.uint8  
torch.Size([16, 1, 128, 192]) torch.uint8
```

How it works...

In the *Data exploration* subsection, we inspected the data to understand more about it. First, we got the list of images and annotations in the `training_set` folder. The annotations files have the term `Annotation` in the filename. As we previously learned, there were 999 images and 999 annotation files in the folder.

In step 2, we displayed a few random images and annotations. We defined a helper function, `show_img_mask`, to show an image and an overlay. The inputs to the helper function were as follows:

- `image`: A `PIL` image
- `mask`: A `PIL` or `numpy` array containing the binary mask of the object

In the function, we used `mark_boundaries` from the `skimage.segmentation` package to overlay the mask on the image.



Note that the annotation files contain the edges of the fetal head. As such, we converted the edges to binary masks using the `binary_fill_holes` function from the `scipy.ndimage` package. The screenshot displayed original ultrasound images on the left, binary masks of the fetal heads in the middle, and the image overlay on the right.

In the *Data augmentation* subsection, we used a popular Python package called `albumentations` for data transformation. For the first time, you needed to install the package, as shown in step 1. Make sure that you install the package in the `conda` environment that we created for this book. The package has a rich set of various augmentation and transformation techniques. In step 2, we imported a few of these packages. Next, for the training dataset, we composed a vertical and horizontal flip, resizing the images to 128×192 . For the validation dataset, we only performed the resizing transformation. We will pass these transformations to the PyTorch dataset class in the next subsection.

In *Creating the dataset class* subsection, we used the `Dataset` class from the `torch.utils.data` package to create custom training and validation datasets. In step 1, we defined the `fetal_dataset` class. The class has three functions. The first function is the `__init__` function with three inputs, which were as follows:

- `self`: Pointer to the class
- `path2data`: A string, the location of the data
- `transform`: The transformation function

In the function, we got the list of images and annotations and initialized the variables.

The second function of the class, `__len__`, returns the length of the dataset. Finally, the `__getitem__` function loads the image and annotation files and returns the image and binary mask. Note that we converted the arrays to PyTorch tensors using the `to_tensor` function at the end. The `to_tensor` function normalizes the values by diving them to 255. As such, we scaled back the mask values to the range of $[0, 1]$ by multiplying it by 255.

In step 2, we defined two objects of the `fetal_dataset` class by passing `transform_train` and `transform_val` to the class, respectively. As expected, the length of both `fetal_ds1` and `fetal_ds2` is 999.

In step 3, we fetched and displayed an image and a mask from `fetal_ds1`. Note the shape, type, and maximum value in the image and mask. As you can see, the image and mask have been resized to 128×192 , which are of the `torch.FloatTensor` type, and have a maximum value of 1.

In step 4, we created two lists of `train_index` and `val_index` by separating 20 percent of the indices from the total indices.

In step 5, we passed the `train_index` and `val_index` to the `Subset` class from the `torch.utils.data` package to create the training and validation datasets, respectively.

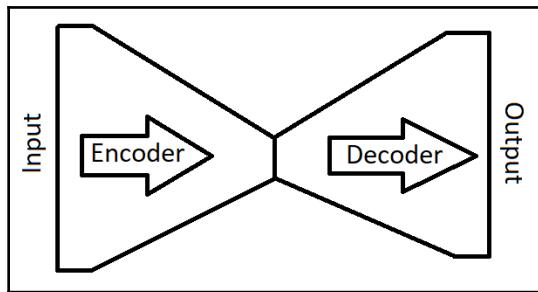
In step 6, we displayed the sample image and mask from `train_ds` and `val_ds`, respectively.

In step 7, we defined two data loaders, `train_dl`, and `val_dl`, using the `DataLoader` class to be able to fetch data batches from the training and validation datasets during training.

In step 8, we got a data batch from `train_dl` and `val_dl`, and printed their respective shapes. As we saw, tensors of size `[batch_size, 128, 192]` were returned from the data loaders.

Defining the model

The popular model architecture for segmentation tasks is the so-called **encoder–decoder** model, as shown in the following screenshot:



In the first half of the encoder–decoder model, the input image is downsized to a feature map using a few layers of **convolutional neural networks (CNNs)** and pooling layers. In the second half of the model, the feature map is up-sampled to the input image size to produce a binary mask. The encoder–decoder model was further updated based on the concept of skip-connections from ResNet to another popular architecture called U-Net. In this recipe, we will learn how to develop an encoder–decoder model for single-object image segmentation using PyTorch.

How to do it...

We will first define the model class and print the model.

1. Let's define the model class. First, import the required packages:

```
import torch.nn as nn
import torch.nn.functional as F
```

Then, define the SegNet class:

```
class SegNet(nn.Module):
    def __init__(self, params):
        super(SegNet, self).__init__()
        C_in, H_in, W_in=params["input_shape"]
        init_f=params["initial_filters"]
        num_outputs=params["num_outputs"]

        self.conv1 = nn.Conv2d(C_in, init_f,
kernel_size=3,stride=1,padding=1)
        self.conv2 = nn.Conv2d(init_f, 2*init_f,
kernel_size=3,stride=1,padding=1)
        self.conv3 = nn.Conv2d(2*init_f, 4*init_f,
kernel_size=3,padding=1)
        self.conv4 = nn.Conv2d(4*init_f, 8*init_f,
kernel_size=3,padding=1)
        self.conv5 = nn.Conv2d(8*init_f, 16*init_f,
kernel_size=3,padding=1)
```

The SegNet class will continue:

```
        self.upsample = nn.Upsample(scale_factor=2,
mode='bilinear', align_corners=True)

        self.conv_up1 = nn.Conv2d(16*init_f, 8*init_f,
kernel_size=3,padding=1)
        self.conv_up2 = nn.Conv2d(8*init_f, 4*init_f,
kernel_size=3,padding=1)
        self.conv_up3 = nn.Conv2d(4*init_f, 2*init_f,
kernel_size=3,padding=1)
        self.conv_up4 = nn.Conv2d(2*init_f, init_f,
kernel_size=3,padding=1)

        self.conv_out = nn.Conv2d(init_f, num_outputs ,
kernel_size=3,padding=1)
```

The SegNet class will continue by defining the forward function:

```
def forward(self, x):
    x = F.relu(self.conv1(x))
    x = F.max_pool2d(x, 2, 2)

    x = F.relu(self.conv2(x))
    x = F.max_pool2d(x, 2, 2)

    x = F.relu(self.conv3(x))
    x = F.max_pool2d(x, 2, 2)

    x = F.relu(self.conv4(x))
    x = F.max_pool2d(x, 2, 2)

    x = F.relu(self.conv5(x))
```

The forward function of the SegNet class will continue:

```
x=self.upsample(x)
x = F.relu(self.conv_up1(x))

x=self.upsample(x)
x = F.relu(self.conv_up2(x))
x=self.upsample(x)
x = F.relu(self.conv_up3(x))
x=self.upsample(x)
x = F.relu(self.conv_up4(x))

x = self.conv_out(x)
return x
```

2. Define an object of the SegNet class:

```
params_model={
    "input_shape": (1,h,w),
    "initial_filters": 16,
    "num_outputs": 1,
}

model = SegNet(params_model)
```

3. Move the model to the GPU device, if available:

```
import torch
device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')
model=model.to(device)
```

4. Print the model:

```
print(model)
```

The preceding code snippet will print the following output:

```
SegNet (
  (conv1): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1),
  padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1),
  padding=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1),
  padding=(1, 1))
  (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1),
  padding=(1, 1))
  (conv5): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1),
  padding=(1, 1))
  (upsample): Upsample(scale_factor=2.0, mode=bilinear)
  (conv_up1): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1),
  padding=(1, 1))
  (conv_up2): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1),
  padding=(1, 1))
  (conv_up3): Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1),
  padding=(1, 1))
  (conv_up4): Conv2d(32, 16, kernel_size=(3, 3), stride=(1, 1),
  padding=(1, 1))
  (conv_out): Conv2d(16, 1, kernel_size=(3, 3), stride=(1, 1),
  padding=(1, 1))
)
```

5. Show the model summary:

```
from torchsummary import summary
summary(model, input_size=(1, h, w), device=device.type)
```

The preceding code snippet will print the following output:

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 16, 128, 192]	160
Conv2d-2	[-1, 32, 64, 96]	4,640
Conv2d-3	[-1, 64, 32, 48]	18,496
Conv2d-4	[-1, 128, 16, 24]	73,856
Conv2d-5	[-1, 256, 8, 12]	295,168
Upsample-6	[-1, 256, 16, 24]	0
Conv2d-7	[-1, 128, 16, 24]	295,040
Upsample-8	[-1, 128, 32, 48]	0

Conv2d-9	<code>[-1, 64, 32, 48]</code>	73,792
Upsample-10	<code>[-1, 64, 64, 96]</code>	0
Conv2d-11	<code>[-1, 32, 64, 96]</code>	18,464
Upsample-12	<code>[-1, 32, 128, 192]</code>	0
Conv2d-13	<code>[-1, 16, 128, 192]</code>	4,624
Conv2d-14	<code>[-1, 1, 128, 192]</code>	145

In the next section, we will describe how each step works.

How it works...

In step 1, we defined the model class, `SegNet`. We first imported the required packages. The model class was built of the `__init__` and `forward` functions. The input to this function is a Python dictionary that contains the model parameters:

- `params`: A Python dictionary containing the model parameters

The dictionary keys are as follows:

- `input_shape`: A tuple, the model input shape (`1, height, width`)
- `initial_filters`: An integer, the number of filters in the first CNN layer
- `num_outputs`: An integer, the number of output channels; pass `1` for single-object segmentation

In the `__init__` function, we defined the building blocks of the model. This included five CNN blocks for down-sampling the input, an up-sampling block, and another five CNN blocks for up-sampling the feature map.

In the `forward` function, we defined the connection between the layers. The input to the `forward` function is a tensor with a `(batch_size, 1, height, width)` shape. As we saw, the input image goes through the CNN and max-pooling blocks and is then up-sampled to the input image size.



Note that no activation function was applied to the output layer in the model definition.

In step 2, we defined an object of the `SegNet` class and called it `model`. In step 3, we moved the model to the GPU device if the device was available. In step 4, we printed the model. In step 5, we printed the model summary using the `torchsummary` package. Note that `torchsummary` is not a built-in PyTorch package and you need to separately install it (if you haven't already) in the `conda` environment, as explained in Chapter 1, *Getting Started with PyTorch for Deep Learning*.

Defining the loss function and optimizer

So far, we have created a dataset and a model. To train the model, we need to define a loss function and an optimizer to update the model parameters based on the gradients of the loss. The classical loss function for single-object segmentation is the binary cross-entropy (BCE) loss function. The BCE loss function compares each pixel of the prediction with that of the ground truth; however, we can combine multiple criteria to improve the overall performance of segmentation tasks. A popular technique is to combine the dice metric with the BCE loss. The dice metric is commonly used to test the performance of segmentation algorithms by calculating the amount of overlap between the ground truth and the prediction. In this section, you will learn how to develop a combined loss function. Then, you will define the optimizer to automatically update the model parameters during training.

How to do it...

We will develop the combined loss function, an optimizer, and a learning rate schedule:

1. Define a helper function to calculate the dice metric:

```
def dice_loss(pred, target, smooth = 1e-5):  
  
    intersection = (pred * target).sum(dim=(2,3))  
    union= pred.sum(dim=(2,3)) + target.sum(dim=(2,3))  
    dice= 2.0 * (intersection + smooth) / (union+ smooth)  
    loss = 1.0 - dice  
    return loss.sum(), dice.sum()
```

2. Define a helper function to calculate the combined loss per data batch:

```
import torch.nn.functional as F

def loss_func(pred, target):
    bce = F.binary_cross_entropy_with_logits(pred, target,
reduction='sum')
    pred= torch.sigmoid(pred)
    dlv, _ = dice_loss(pred, target)
    loss = bce + dlv

    return loss
```

3. Define the metrics_batch helper function:

```
def metrics_batch(pred, target):
    pred= torch.sigmoid(pred)
    _, metric=dice_loss(pred, target)
    return metric
```

4. Let's define the loss_batch helper function:

```
def loss_batch(loss_func, output, target, opt=None):
    loss = loss_func(output, target)
    _, metric_b=dice_loss(pred, target)
    if opt is not None:
        opt.zero_grad()
        loss.backward()
        opt.step()

    return loss.item(), metric_b
```

5. Define the optimizer:

```
from torch import optim
opt = optim.Adam(model.parameters(), lr=3e-4)
```

6. Define the learning rate schedule:

```
from torch.optim.lr_scheduler import ReduceLROnPlateau
lr_scheduler = ReduceLROnPlateau(opt, mode='min', factor=0.5,
patience=20, verbose=1)
```

In the next section, we will describe how each step works.

How it works...

In step 1, we defined the `dice_loss` helper function to calculate the dice loss value. The inputs to the function were as follows:

- `pred`: A tensor with a `(batch_size, 1, height, width)` shape, corresponding to predictions
- `target`: A tensor with a `(batch_size, 1, height, width)` shape, corresponding to the ground truth

The function calculates the dice value per data batch. The dice value is between `[0, 1]`, where a value of 1 represents a perfect overlap between the prediction and the ground truth. The dice loss is calculated as `(1 - dice)`, since we wanted to minimize the value. The function returns the sum of loss values per data batch.

In step 2, we defined the `loss_func` helper function to compute the combined loss value per data batch. The function has two inputs:

- `pred`: A tensor with a `(batch_size, 1, height, width)` shape, corresponding to predictions
- `target`: A tensor with a `(batch_size, 1, height, width)` shape, corresponding to the ground truth

In the function, we first calculated the binary cross-entropy loss. Next, we calculated the dice loss and returned the sum of the two losses per data batch.



Note that the `sigmoid` operation is integrated into the `binary_crossentropy_with_logits` function. Remember that the model output does not include the `sigmoid` activation function.

In step 3, we defined the `metrics_batch` helper function to compute a metric per batch. You can calculate any metric you want inside this helper function. We calculated the dice metric by calling the `dice_loss` helper function. The dice metric is a popular metric for evaluating segmentation tasks.

In step 4, we defined the `loss_batch` helper function. The inputs to the helper function are as follows:

- `loss_func`: Combined loss function defined in step 2
- `output`: A tensor with a `(batch_size, 1, height, width)` shape containing predictions
- `target`: A tensor with a `(batch_size, 1, height, width)` shape containing the ground truth
- `opt`: An object of the optimizer

In the function, we calculated the loss and metric values per data batch. During training, the optimizer object is passed to the helper function, and, as a result, the model parameters are updated using `opt.step()`.

In step 5, we defined the Adam optimizer to optimize the model during training.

In step 6, we defined the learning rate schedule to automatically reduce the learning rate during training in case of a plateau.

Training the model

So far, we have learned how to create the training and validation datasets, build the model, and define the loss function and optimizer. At this point, it is time to train the model. This is an iterative process. In each iteration, we select a data batch from the training dataset. We then feed the data to the model to get the model output. Then, we calculate the loss value. Next, we compute the gradients of the loss function with respect to the model parameters (also known as weights). Finally, the optimizer updates the parameters based on the gradients, and this loop continues. We also use the validation dataset to monitor the model performance during training. We will stop the training process when the performance plateaus. For better code readability, we will define a few helper functions.

How to do it...

We will first train and evaluate the model:

1. Define the `loss_epoch` helper function:

```
def loss_epoch(model, loss_func, dataset_dl, sanity_check=False, opt=None):
    running_loss=0.0
    running_metric=0.0
    len_data=len(dataset_dl.dataset)

    for xb, yb in dataset_dl:
        xb=xb.unsqueeze(1).type(torch.float32).to(device)
        yb=yb.unsqueeze(1).type(torch.float32).to(device)
        output=model(xb)
        loss_b, metric_b=loss_batch(loss_func, output, yb, opt)
        running_loss += loss_b
```

The `loss_epoch` helper function will continue:

```
    if metric_b is not None:
        running_metric+=metric_b
    if sanity_check is True:
        break
    loss=running_loss/float(len_data)
    metric=running_metric/float(len_data)
    return loss, metric
```

2. Define the `train_val` helper function:

```
import copy
def train_val(model, params):
    num_epochs=params["num_epochs"]
    loss_func=params["loss_func"]
    opt=params["optimizer"]
    train_dl=params["train_dl"]
    val_dl=params["val_dl"]
    sanity_check=params["sanity_check"]
    lr_scheduler=params["lr_scheduler"]
    path2weights=params["path2weights"]
```

The `train_val` helper function will continue:

```
    loss_history={
        "train": [],
        "val": []
    }
    metric_history={
```

```

        "train": [],
        "val": []
    best_model_wts = copy.deepcopy(model.state_dict())
    best_loss=float('inf')

```

The `train_val` helper function will continue:

```

    for epoch in range(num_epochs):
        current_lr=get_lr(opt)
        print('Epoch {} / {}, current lr={}'.format(epoch, num_epochs - 1, current_lr))

        model.train()
        train_loss,
        train_metric=loss_epoch(model, loss_func, train_dl, sanity_check, opt)

        loss_history["train"].append(train_loss)
        metric_history["train"].append(train_metric)

```

The `train_val` helper function will continue:

```

        model.eval()
        with torch.no_grad():
            val_loss,
        val_metric=loss_epoch(model, loss_func, val_dl, sanity_check)
        loss_history["val"].append(val_loss)
        metric_history["val"].append(val_metric)
        if val_loss < best_loss:
            best_loss = val_loss
            best_model_wts = copy.deepcopy(model.state_dict())
            torch.save(model.state_dict(), path2weights)
            print("Copied best model weights!")

```

The `train_val` helper will continue:

```

        lr_scheduler.step(val_loss)
        if current_lr != get_lr(opt):
            print("Loading best model weights!")
            model.load_state_dict(best_model_wts)
            print("train loss: %.6f, dice: %.2f" % (train_loss, 100 * train_metric))
            print("val loss: %.6f, dice: %.2f" % (val_loss, 100 * val_metric))
            print("-" * 10)

        model.load_state_dict(best_model_wts)
        return model, loss_history, metric_history

```

3. Now, let's call the `train_val` function to train the model:

```
path2models= "./models/"
if not os.path.exists(path2models):
    os.mkdir(path2models)

params_train={
    "num_epochs": 100,
    "optimizer": opt,
    "loss_func": loss_func,
    "train_dl": train_dl,
    "val_dl": val_dl,
    "sanity_check": False,
    "lr_scheduler": lr_scheduler,
    "path2weights": path2models+"weights.pt",
}
model, loss_hist, metric_hist=train_val(model, params_train)
```

The training process will start and print the following output:

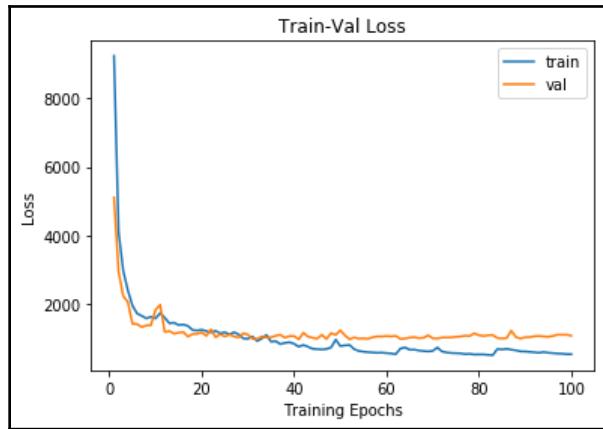
```
Epoch 0/99, current lr=0.0003
Copied best model weights!
train loss: 8438.457021, accuracy: 61.10
val loss: 4836.094668, accuracy: 79.14
-----
Epoch 1/99, current lr=0.0003
Copied best model weights!
train loss: 4296.525105, accuracy: 78.74
val loss: 4189.101631, accuracy: 82.83
...
```

4. Let's plot the progress of the training and validation losses:

```
num_epochs=params_train["num_epochs"]

plt.title("Train-Val Loss")
plt.plot(range(1,num_epochs+1),loss_hist["train"],label="train")
plt.plot(range(1,num_epochs+1),loss_hist["val"],label="val")
plt.ylabel("Loss")
plt.xlabel("Training Epochs")
plt.legend()
plt.show()
```

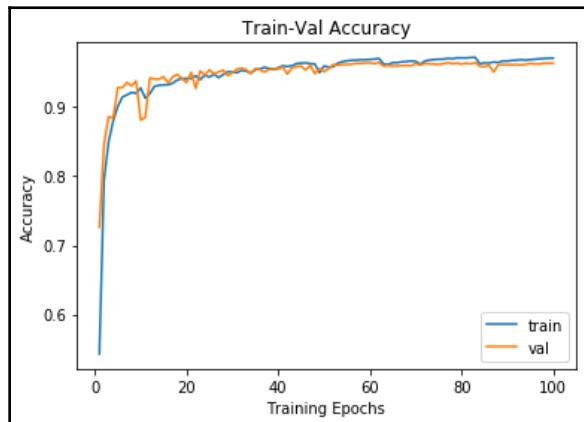
The preceding snippet will plot the following graph:



It will also plot the progress of the dice values:

```
plt.title("Train-Val Accuracy")
plt.plot(range(1,num_epochs+1),metric_hist["train"],label="train")
plt.plot(range(1,num_epochs+1),metric_hist["val"],label="val")
plt.ylabel("Accuracy")
plt.xlabel("Training Epochs")
plt.legend()
plt.show()
```

The preceding snippet will plot the following graph:



In the next section, we will describe how each step works.

How it works...

In step 1, we defined the `loss_epoch` helper function. The function inputs are as follows:

- `model`: An object of the model
- `loss_func`: An object of the loss function
- `dataset_dl`: An object of the data loader
- `sanity_check`: A Boolean flag; default value is `False`
- `opt`: An object of the optimizer

In the function, we extracted data batches from the data loader as `xb` and `yb` tensors. Next, we obtained the model output and calculated the loss and metric values per data batch. We repeated the process for the entire dataset and returned the average loss and metric values.

In step 2, we defined the `train_val` helper function. The inputs to the function are as follows:

- `model`: An object of the mode
- `params`: A Python dictionary containing the training parameters

In the function, we trained the model for `num_epochs` iterations. In each iteration, we set the model in `train` mode and trained the model for an epoch. Then we evaluated the model on the validation dataset. We stored the model parameters if the validation results were improved in each iteration. We also used the learning rate schedule to reduce the learning rate if the validation performance reached a plateau. The function returned the trained model and two dictionaries containing the loss and metric values for each iteration.

In step 3, we set the training parameters in `params_train` and called the `train_val` function to train the model.

In step 4, we plotted the results of the training and validation loss values to see the progress of training over multiple epochs.

Deploying the model

Once the training is complete, you will want to deploy the model on new data. In this recipe, you will learn how to deploy the model on the `test_set` images. We will assume that you want to deploy the model for inference in a new script separate from the training scripts.

In this case, the model does not exist in memory. To avoid repetition, we will skip the model definition. Make sure that you define the model in your deployment code as explained in the *Defining the model* section before you go through the following steps.

How to do it...

We will load a few images from the `test_set` dataset, feed them to the trained model, and then display the outputs:

1. Get a list of images in the `test_set` folder:

```
import os
path2test="../data/test_set/"
imgsList=[pp for pp in os.listdir(path2test) if "Annotation" not in pp]
print("number of images:", len(imgsList))
```

The preceding code snippet will print the following output:

```
number of images: 335
```

2. Get a few random images:

```
import numpy as np
np.random.seed(2019)
rndImgs=np.random.choice(imgsList,4)
rndImgs
```

The preceding code snippet will print the following output:

```
array(['043_HC.png', '011_HC.png', '303_HC.png', '091_HC.png'],
      dtype='<U10')
```

3. Load the trained weights into the model:

```
path2weights="../models/weights.pt"
model.load_state_dict(torch.load(path2weights))
model.eval()
```

4. Load the images and feed them to the model:

```
from torchvision.transforms.functional import to_tensor,
to_pil_image

for fn in rndImgs:
    path2img = os.path.join(path2train, fn)
```

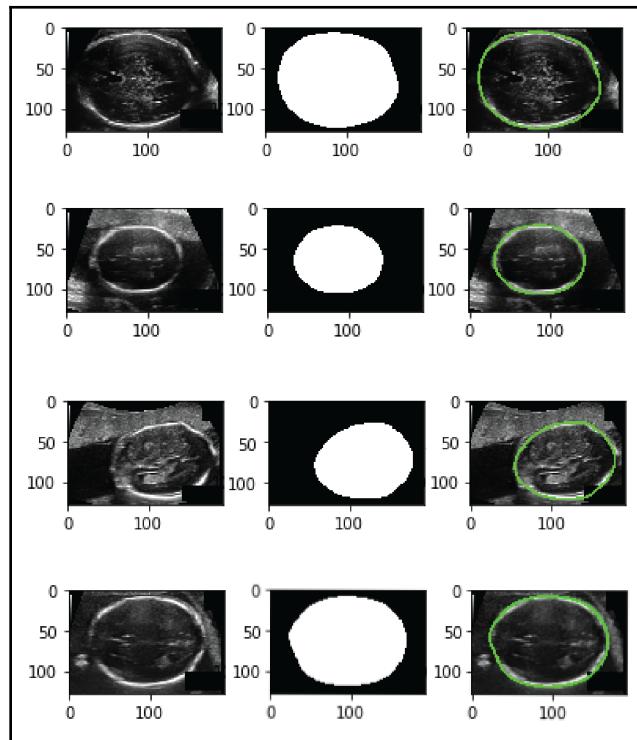
```
img = Image.open(path2img)
img=img.resize((w,h))
img_t=to_tensor(img).unsqueeze(0).to(device)
pred=model(img_t)
pred=torch.sigmoid(pred)[0]
mask_pred= (pred[0]>=0.5)
```

Display the image and the predicted output:

```
plt.figure()
plt.subplot(1, 3, 1)
plt.imshow(img, cmap="gray")

plt.subplot(1, 3, 2)
plt.imshow(mask_pred, cmap="gray")
plt.subplot(1, 3, 3)
show_img_mask(img, mask_pred)
```

The preceding code snippet will display the following image:



In the next section, we will describe how each step works.

How it works...

In step 1, we got the list of images in the `test_set` folder. As you saw, there were 335 `png` images. There were no annotation files for the test set.

In step 2, we picked four random images.

In step 3, we loaded the trained weights into the model. The model was set in the evaluation mode using the `eval` method.

In step 4, we loaded the images one by one and fed them to the model. Before we fed an image to the model, we resized it to `(128, 192)` and converted it to a PyTorch tensor. The model output was then passed to the `sigmoid` activation function and compared to a `0.5` threshold to get a binary mask. Next, we displayed the image, predicted the binary mask, and predicted the contours that were overlaid on the image. As you saw, the model was able to accurately detect the fetal head in ultrasound images.

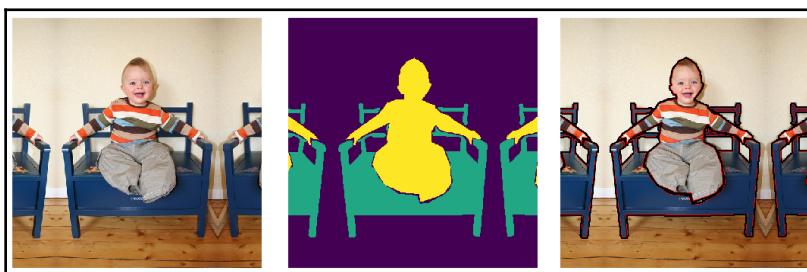
7

Multi-Object Segmentation

In object segmentation tasks, we aim to find the boundaries of target objects in images. There are many applications for segmenting objects in images, such as self-driving cars and medical imaging analysis. In [Chapter 6, Single-Object Segmentation](#), we learned how to build a single-object segmentation model using PyTorch. In this chapter, we will focus on developing a deep learning model using PyTorch to perform multi-object segmentation. In multi-object segmentation, we are interested in automatically outlining the boundaries of multiple target objects in an image.

The boundaries of objects in an image are usually defined by a segmentation mask that's the same size as the image. In the segmentation mask, all the pixels that belong to a target object are labeled the same based on a pre-defined labeling. For instance, in the following screenshot, you can see a sample image with two types of target: babies and chairs. The corresponding segmentation mask is shown in the middle of the screenshot. As we can see, the pixels belonging to the babies and chairs are labeled differently and colored in yellow and green, respectively.

From the segmentation mask, we can overlay boundary contours on the image, as shown on the right-hand side of the screenshot:



The goal of multi-object segmentation is to predict a segmentation mask when given an image so that each pixel in the image is labeled based on its object class. In this chapter, we will learn how to develop an algorithm to automatically segment 20 target objects in the **Visual Object Classes (VOC)** dataset.

In this chapter, we will cover the following recipes:

- Creating custom datasets
- Defining and deploying a model
- Defining the loss function and optimizer
- Training the model

Creating custom datasets

We will be using the VOC segmentation data from the `torchvision.datasets` package. This dataset consists of outlined images with 20 object classes. Together with the background, this comes to 21 object classes, which were labeled from 0 to 20:

- background: 0
- airplane: 1
- bicycle: 2
- bird: 3
- boat: 4
- bottle: 5
- bus: 6
- car: 7
- cat: 8
- chair: 9
- cow: 10
- dining table: 11
- dog: 12
- horse: 13
- motorbike: 14
- person: 15
- potted plant: 16
- sheep: 17
- sofa: 18
- train: 19
- tv/monitor : 20



To learn more about the VOC dataset, visit the following link: <http://host.robots.ox.ac.uk/pascal/VOC/voc2012/index.html>.

In this recipe, we will learn how to load the `VOCSegmentation` data using the `torchvision` package and create custom dataset classes that will be used to train and validate the model.

How to do it...

In this recipe, we will create a custom dataset class and define data loaders. Let's get started:

1. Import the required packages:

```
from torchvision.datasets import VOCSegmentation
from torchvision.transforms.functional import to_tensor,
to_pil_image
from PIL import Image
```

2. Define a custom dataset class by subclassing the `VOCSegmentation` class:

```
class myVOCSegmentation(VOCSegmentation):
    def __getitem__(self, index):
        img = Image.open(self.images[index]).convert('RGB')
        target = Image.open(self.masks[index])

        if self.transforms is not None:
            augmented= self.transforms(image=np.array(img),
mask=np.array(target))
            img = augmented['image']
            target = augmented['mask']
            target[target>20]=0

        img= to_tensor(img)
        target= torch.from_numpy(target).type(torch.long)
        return img, target
```

3. Define transformation functions:

```
from albumentations import (
    HorizontalFlip,
    Compose,
    Resize,
```

```
Normalize)

mean = [0.485, 0.456, 0.406]
std = [0.229, 0.224, 0.225]
h,w=520,520

transform_train = Compose([ Resize(h,w),
                           HorizontalFlip(p=0.5),
                           Normalize(mean=mean, std=std) ])

transform_val = Compose( [ Resize(h,w),
                           Normalize(mean=mean, std=std) ])
```

4. Define an object of the `myVOCSegmentation` class for training:

```
path2data="./data/"
train_ds=myVOCSegmentation( path2data,
                            year='2012',
                            image_set='train',
                            download=True,
                            transforms=transform_train)
print(len(train_ds))
```

The preceding snippet will print the following output:

```
1464
```

Define an object of the `myVOCSegmentation` class for validation:

```
val_ds=myVOCSegmentation(path2data,
                          year='2012',
                          image_set='val',
                          download=True,
                          transforms=transform_val)
print(len(val_ds))
```

The preceding snippet will print the following output:

```
1449
```

5. Let's show a sample image and its segmentation mask from the training data.

Import the required packages:

```
import torch
import numpy as np
from skimage.segmentation import mark_boundaries
import matplotlib.pyplot as plt
%matplotlib inline
```

```
np.random.seed(0)
num_classes=21
COLORS = np.random.randint(0, 2, size=(num_classes+1,
3), dtype="uint8")
```

Define the `show_img_target` helper function:

```
def show_img_target(img, target):
    if torch.is_tensor(img):
        img=to_pil_image(img)
        target=target.numpy()
    for ll in range(num_classes):
        mask=(target==ll)
        img=mark_boundaries(np.array(img) ,
                           mask,
                           outline_color=COLORS[ll],
                           color=COLORS[ll])
    plt.imshow(img)
```

Define the `re_normalize` helper function:

```
def re_normalize (x, mean = mean, std= std):
    x_r= x.clone()
    for c, (mean_c, std_c) in enumerate(zip(mean, std)):
        x_r [c] *= std_c
        x_r [c] += mean_c
    return x_r
```

Get a sample image and its segmentation mask from `train_ds`:

```
img, mask = train_ds[6]
print(img.shape, img.type(), torch.max(img))
print(mask.shape, mask.type(), torch.max(mask))
```

The preceding snippet will print the following output:

```
torch.Size([3, 520, 520]) torch.FloatTensor tensor(2.6400)
torch.Size([520, 520]) torch.LongTensor tensor(4)
```

Display the sample image with its segmentation mask and contours:

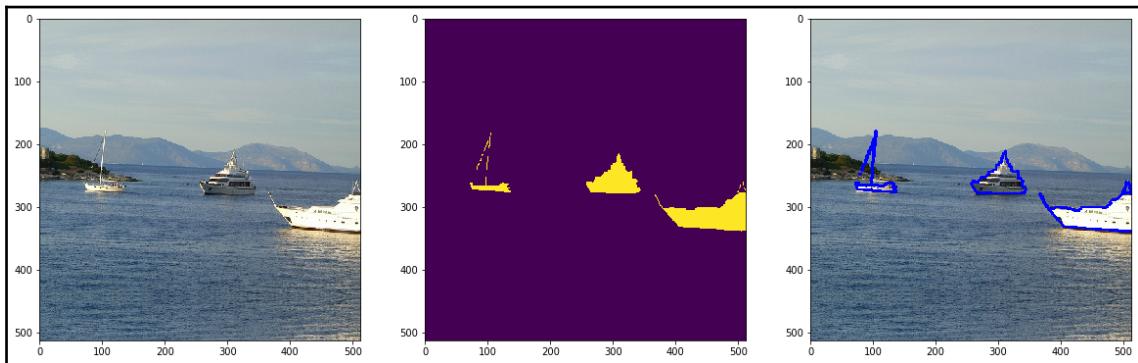
```
plt.figure(figsize=(20,20))

img_r= re_normalize(img)
plt.subplot(1, 3, 1)
plt.imshow(to_pil_image(img_r))

plt.subplot(1, 3, 2)
plt.imshow(mask)
```

```
plt.subplot(1, 3, 3)
show_img_target(img_r, mask)
```

The preceding snippet will display the following screenshot:



6. Similarly, get a sample image with its segmentation mask from val_ds:

```
img, mask = val_ds[0]
print(img.shape, img.type(), torch.max(img))
print(mask.shape, mask.type(), torch.max(mask))
```

The preceding snippet will print the following output:

```
torch.Size([3, 520, 520]) torch.FloatTensor tensor(2.6226)
torch.Size([520, 520]) torch.LongTensor tensor(1)
```

Display the sample image with its segmentation mask and contours:

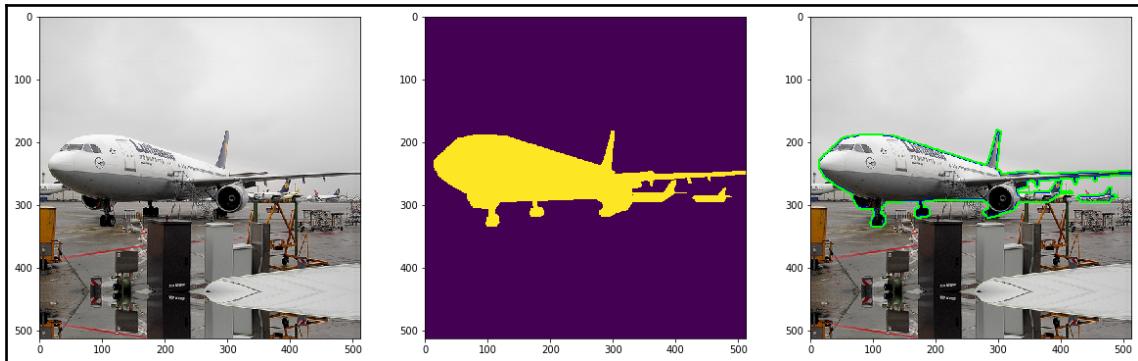
```
plt.figure(figsize=(20,20))

img_r= re_normalize(img)
plt.subplot(1, 3, 1)
plt.imshow(to_pil_image(img_r))

plt.subplot(1, 3, 2)
plt.imshow(mask)

plt.subplot(1, 3, 3)
show_img_target(img_r, mask)
```

The preceding snippet will display the following screenshot:



7. Define data loaders:

```
from torch.utils.data import DataLoader
train_dl = DataLoader(train_ds, batch_size=4, shuffle=True)
val_dl = DataLoader(val_ds, batch_size=8, shuffle=False)
```

8. Get a sample data batch from train_dl:

```
for img_b, mask_b in train_dl:
    print(img_b.shape, img_b.dtype)
    print(mask_b.shape, mask_b.dtype)
    break
```

The preceding snippet will print the following output:

```
torch.Size([4, 3, 520, 520]) torch.float32
torch.Size([4, 520, 520]) torch.int64
```

Get a sample data batch from val_dl:

```
for img_b, mask_b in val_dl:
    print(img_b.shape, img_b.dtype)
    print(mask_b.shape, mask_b.dtype)
    break
```

The preceding snippet will print the following output:

```
torch.Size([8, 3, 520, 520]) torch.float32
torch.Size([8, 520, 520]) torch.int64
```

In the next section, we will describe how each step works.

How it works...

In *step 1*, we imported the required packages. We imported `VOCSegmentation` from the `torchvision.datasets` package so that we can load the VOC dataset. We also imported the `to_tensor` and `to_pil_image` methods from `torchvision.transforms.functional` to convert `PIL` images into PyTorch tensors and vice versa.

In *step 2*, we defined a custom dataset class by subclassing the `VOCSegmentation` class. The reason for subclassing is that we needed to make a modification to the `__getitem__` method of the `VOCSegmentation` class. In the method, we loaded the image and target as `PIL` objects from their files. Then, we applied transformations on the image and target.



Note that the pixel values in the target should be in the range of [0, 20] since there are 21 object classes in the VOC dataset. As such, we set any possible value greater than 20 to zero so that it's counted as a background pixel: `target[target > 20] = 0`.

In *step 3*, we defined the transformation function using the `albumentations` package. We learned how to install and use this package in [Chapter 6, Single-Object Segmentation](#). For the training data, three transformations called `Resize`, `HorizontalFlip`, and `Normalize` were applied. For the validation data, two transformations were applied: `Resize` and `Normalize`. We used `Resize` to resize all the images so that they were the same size. We used `HorizontalFlip` to augment the training data by flipping the images horizontally. Finally, we used `Normalize` to normalize the images.

In *step 4*, we defined two objects of the `myVOCSegmentation` class for the training and validation datasets. The arguments for this class are as follows:

- `path2data`: A string. The path to store/load the VOC dataset after downloading it
- `year`: The dataset year. A '2012' or '2007' string
- `image_set`: Set to 'train' for training and 'val' for validation
- `download`: Set to `True` to download the data from the internet for the first time
- `transforms`: Augmentation functions

As we saw, there were 1446 and 1449 items in `train_ds` and `val_ds`, respectively.

In *step 5*, we got a sample image and its target mask from `train_ds`. Then, we printed the shape of the tensor and displayed the image and mask. Note that the returned image is a tensor of the `torch.FloatTensor` type with a maximum pixel value of 2.64.



The `to_tensor` method converts a PIL image into a PyTorch tensor with pixel values in the range of [0, 1]. However, due to the normalization function, the maximum pixel values will exceed 1.

Also, the returned mask was a tensor of the `torch.LongTensor` type with a maximum value of 4.



The image contains boats, which were assigned label 4 in the VOC dataset.

To display the image, we renormalize the image using the `re_normalize` helper function. The inputs to the function are as follows:

- `x`: A normalized tensor of shape [3, Height, Width].
- `mean`: A tuple/list of three values. The default value is [0.485, 0.456, 0.406].
- `std`: A tuple/list of three values. The default value is [0.229, 0.224, 0.225].

As you may recall, images are normalized in `myVOCSegmentation` using the `normalize` transformation function. Using the `re_normalize` helper function, we get the original pixel values of the image for display purposes only.

To display the image and its target mask, we defined a helper function called `show_img_target`. The helper function's inputs are as follows:

- `img`: A PIL image or PyTorch tensor
- `target`: A NumPy array or PyTorch tensor

In this function, we used the `mark_boundaries` function from the `skimage.segmentation` package to overlay a mask on the image. Note that, if you rerun this part of the code, the image and its mask will be flipped due to the `HorizontalFlip` transformation.

In step 6, we got a sample image and its target mask from `val_ds`, printed their shape, and displayed them. As you saw, the returned mask was a tensor of integer values with a maximum value of 1.



The image contains an airplane, which was assigned label 1 in the VOC dataset.

In *step 7*, we defined the `train_dl` and `val_dl` data loaders for the training and validation datasets, respectively.

In *step 8*, we extracted data batches from `train_dl` and `val_dl`, respectively. Note that the shapes of the returned tensors are `(batch_size, 3, height, width)` and `(batch_size, height, width)`, respectively.

Defining and deploying a model

In this recipe, we will learn how to create and deploy a semantic segmentation model using the `torchvision.models` package. The model is called **DeepLabV3Net101** and is based on the article *Rethinking Atrous Convolution for Semantic Image Segmentation* (<https://arxiv.org/abs/1706.05587>).

This article emphasizes the use of atrous convolutions to overcome the limitations of regular convolutional neural networks and pooling layers in semantic segmentation.

The model was pre-trained on a subset of the **COCO train2017 dataset**. When we import the model, we have the option to load pre-trained weights or use random weights.

The model expects input images to be scaled to the range of `[0, 1]` and then normalized by subtracting `mean=[0.485, 0.456, 0.406]` and dividing by `std= [0.229, 0.224, 0.225]`. Also, the model was trained on images that were resized to 520 by 520. As you may recall from the *Creating custom datasets* recipe, we applied all these transformations to the inputs in the `myVOCSegmentation` class. Therefore, once a data batch has been fetched from the data loader, it is ready for the model.

Since the model was pre-trained using the COCO train2017 dataset, we can deploy the model right away on the VOC dataset. However, to get better performance, we can also train the model on the VOC train dataset before deploying it. We will learn how to train the model in the *Training the model* recipe.

In this recipe, we will show you how to load the model with pre-trained weights and then deploy it on the VOC validation dataset.

How to do it...

In this recipe, we will define the segmentation model and print it. Let's get started:

1. Import the required packages:

```
import torch
from torchvision.models.segmentation import deeplabv3_resnet101
```

2. Define an object of the segmentation class:

```
model=deeplabv3_resnet101(pretrained=True, num_classes=21)
```

3. Move the model to the CUDA device:

```
device = torch.device('cuda:0' if torch.cuda.is_available() else
'cpu')
model=model.to(device)
```

4. Print the model:

```
print(model)
```

The preceding snippet will print the following output:

```
DeepLabV3(
  (backbone): IntermediateLayerGetter(
    (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (relu): ReLU(inplace=True)
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1,
dilation=1, ceil_mode=False)
    (layer1): Sequential(
      (0): Bottleneck(
        (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1),
bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
      ....
```

5. Deploy the model on a data batch from the validation dataset:

```
from torch import nn

model.eval()
with torch.no_grad():
    for xb, yb in val_dl:
        yb_pred = model(xb.to(device))
        yb_pred = yb_pred["out"].cpu()
        print(yb_pred.shape)
        yb_pred = torch.argmax(yb_pred, axis=1)
        break
print(yb_pred.shape)
```

The preceding snippet will print the following output:

```
torch.Size([8, 21, 520, 520])
torch.Size([8, 520, 520])
```

6. Display a sample image and the corresponding predictions:

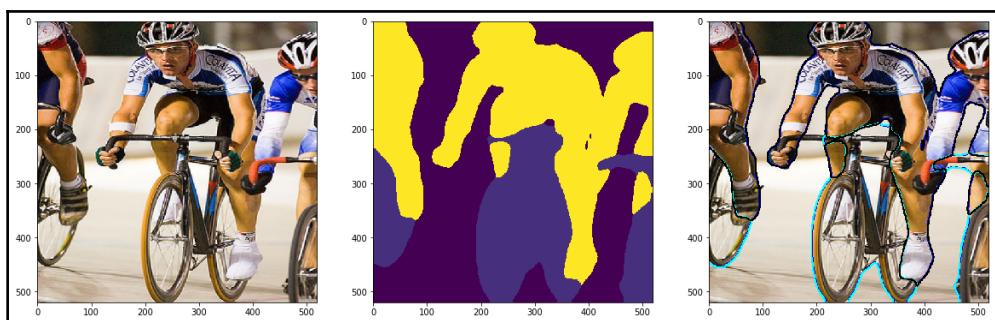
```
plt.figure(figsize=(20,20))

n=4
img, mask= xb[n], yb_pred[n]
img_r= re_normalize(img)
plt.subplot(1, 3, 1)
plt.imshow(to_pil_image(img_r))

plt.subplot(1, 3, 2)
plt.imshow(mask)

plt.subplot(1, 3, 3)
show_img_target(img_r, mask)
```

The preceding snippet will display the following screenshot:



In the next section, we will describe how each step works.

How it works...

In *step 1*, we imported the required packages. The `deeplabv3_resnet101` class is implemented in `torchvision models`.

In *step 2*, we created an object of the `deeplabv3_resnet101` class. The important inputs to the class are as follows:

- `pretrained`: If set to `True`, pre-trained weights are automatically downloaded
- `num_classes`: The number of object classes. The default value is set to 21 for the VOC dataset.

In *step 3*, we defined a CUDA device, if available, and then moved the model to the CUDA device.

In *step 4*, we printed the model. The model is huge, so only a snippet of the model was shown. You should see the entire model on the screen when executing the `print` command on your computer.

In *step 5*, we deployed the model on the VOC validation dataset. As we mentioned previously, since the model was pre-trained on a subset of the COCO 2017 dataset, it should perform well on the VOC dataset. First, we set the model in the evaluation mode. This is necessary since certain layers, such as the dropout and normalization layers, perform differently in the training and evaluation stages.



Before deploying the model, use the `.eval()` method to set the model in the evaluation mode.

Next, we fetched a data batch from the `val_dl` data loader and fed it to the model. The good news is that no pre-processing is required.



To deploy the model on an individual image outside the VOC dataset or images streaming from a camera, make sure to pre-process the image before feeding it to the model.

As we saw, the model output shape is [batch_size, num_classes, height, width]. In other words, the model produces 21 probability masks corresponding to 21 object classes. The first mask at index zero corresponds to the background, while the remaining masks correspond to 20 objects. Then, for each pixel, the class with the highest probability among the 21 masks was assigned to the pixel using the `argmax()` method.

In *step 6*, we displayed a sample image from the extracted data batch, the predicted segmentation mask, and the overlay contours. As we saw, the pre-trained model did a very good job of outlining the objects. You can try displaying other images and predictions by changing the image index, `n`.

See also

You can find other segmentation models, such as `fcn_resnet50` in the `torchvision` package. Try them out and see how they perform for this task.

Defining the loss function and optimizer

The classical criterion for multi-object segmentation is the **cross-entropy** (CE) loss function. The CE loss compares each pixel of the predictions with that of the ground truth. We will use the `CrossEntropyLoss` class from the `torch.nn` package in this recipe.

For the optimization, we will use the Adam optimizer from the `torch.optim` package with a small learning rate since the model was pre-trained. If you want to train the model from scratch using random weights, try using a higher learning rate, such as `3E-4`. The learning rate is a hyperparameter. Typically, you will need to try a few different values before finalizing its value for the best performance.

In this recipe, you will learn how to define a loss function, optimizer, and learning rate schedule for multi-object segmentation.

How to do it...

In this recipe, we will develop a loss function, an optimizer, and a learning rate schedule. Let's get started:

1. Define the criterion:

```
from torch import nn
criterion = nn.CrossEntropyLoss(reduction="sum")
```

2. Define the optimizer:

```
from torch import optim
opt = optim.Adam(model.parameters(), lr=1e-6)
```

3. Define a helper function to calculate the loss per data batch:

```
def loss_batch(loss_func, output, target, opt=None):
    loss = loss_func(output, target)
    if opt is not None:
        opt.zero_grad()
        loss.backward()
        opt.step()

    return loss.item(), None
```

4. Define the learning rate schedule:

```
from torch.optim.lr_scheduler import ReduceLROnPlateau
lr_scheduler = ReduceLROnPlateau(opt, mode='min', factor=0.5,
patience=20, verbose=1)
```

5. Define a helper function to get the learning rate:

```
def get_lr(opt):
    for param_group in opt.param_groups:
        return param_group['lr']

current_lr=get_lr(opt)
print('current lr={}'.format(current_lr))
```

The preceding snippet will print the following output:

```
current lr=1e-06
```

In the next section, we will describe how each step works.

How it works...

In *step 1*, we defined the criterion. We used the `CrossEntropyLoss` class from the `torch.nn` package to do so. This criterion combines `nn.LogSoftmax()` and `nn.NLLLoss()` into a single class.



For numerical stability reasons, PyTorch integrates the softmax function into the `CrossEntropyLoss` loss instead of directly applying it to the model output.

Note that we set `reduction="sum"` since we want to return the sum of the loss values per data batch.

In *step 2*, we defined the optimizer. We used the `Adam` optimizer from the `torch.optim` package with a small learning rate since the model was pre-trained. If you want to train the model from scratch using random weights, try using a higher learning rate, such as `3E-4`.

In *step 3*, we defined the `loss_batch` helper function. The inputs to the helper function are as follows:

- `loss_func`: The criterion that was defined in *step 2*
- `output`: A tensor of shape `(batch_size, num_classes, height, width)` containing predictions
- `target`: A tensor of shape `(batch_size, height, width)` containing the ground truth or targets
- `opt`: An object of the optimizer

In this function, we calculated the loss values per data batch. During training, the optimizer object is passed to the helper function and, as a result, the model parameters are updated using `opt.step()`.

In *step 4*, we defined the learning rate schedule to automatically reduce the learning rate during training in the event of a plateau.

In *step 5*, we defined a helper function to read the current learning rate. This will be useful for monitoring the learning rate during training.

Training the model

So far, we've learned how to create training and validation datasets, build a model, and define a loss function and optimizer. We also deployed the model since it was pre-trained on a large dataset called COCO 2017. Sometimes, we will want to train the model on other datasets to get better performance. In such cases, the best method is to train the model with a small learning rate on the new dataset. This method is usually called fine-tuning the model.

Training is an iterative process. In each iteration, we select a data batch from the training dataset. Then, we feed the data to the model to get the model output. After that, we calculate the loss value. Next, we compute the gradients of the loss function with respect to the model parameters (also known as the weights). Finally, the optimizer updates the parameters based on the gradients. This loop continues. We also use the validation dataset to monitor the model's performance during training. We stop the training process when the performance plateaus.

In this recipe, you will learn how to train the model on the VOC dataset for a multi-object segmentation task.

How to do it...

In this recipe, we will train and evaluate a model. Let's get started:

1. Define the `loss_epoch` helper function:

```
def
    loss_epoch(model, loss_func, dataset_dl, sanity_check=False, opt=None):
        running_loss=0.0
        len_data=len(dataset_dl.dataset)
        for xb, yb in dataset_dl:
            xb=xb.to(device)
            yb=yb.to(device)
            output=model(xb) ["out"]
            loss_b, metric_b = loss_batch(loss_func, output, yb, opt)
            running_loss += loss_b
            if sanity_check is True:
                break
        loss=running_loss/float(len_data)
        return loss, None
```

2. Define the `train_val` helper function:

```
import copy
def train_val(model, params):
    num_epochs=params["num_epochs"]
    loss_func=params["loss_func"]
    opt=params["optimizer"]
    train_dl=params["train_dl"]
    val_dl=params["val_dl"]
    sanity_check=params["sanity_check"]
    lr_scheduler=params["lr_scheduler"]
    path2weights=params["path2weights"]
```

The `train_val` helper function continues with the following code:

```
loss_history={
    "train": [],
    "val": []
}
best_model_wts = copy.deepcopy(model.state_dict())
best_loss=float('inf')
```

The `train_val` helper function continues with the following code:

```
for epoch in range(num_epochs):
    current_lr=get_lr(opt)
    print('Epoch {} / {}, current lr={}'.format(epoch, num_epochs - 1, current_lr))

    model.train()
    train_loss, _ =
    loss_epoch(model, loss_func, train_dl, sanity_check, opt)
    loss_history["train"].append(train_loss)
    model.eval()
    with torch.no_grad():
        val_loss, _ =
    loss_epoch(model, loss_func, val_dl, sanity_check)
    loss_history["val"].append(val_loss)
```

The `train_val` helper function continues with the following code:

```
if val_loss < best_loss:  
    best_loss = val_loss  
    best_model_wts = copy.deepcopy(model.state_dict())  
    torch.save(model.state_dict(), path2weights)  
    print("Copied best model weights!")  
    lr_scheduler.step(val_loss)  
    if current_lr != get_lr(opt):  
        print("Loading best model weights!")  
        model.load_state_dict(best_model_wts)
```

The `train_val` helper function continues with the following code:

```
print("train loss: %.6f" %(train_loss))  
print("val loss: %.6f" %(val_loss))  
print("-"*10)  
model.load_state_dict(best_model_wts)  
return model, loss_history, metric_history
```

3. Let's call the `train_val` function to train the model:

```
import os  
path2models= "./models/"  
if not os.path.exists(path2models):  
    os.mkdir(path2models)  
params_train={  
    "num_epochs": 100,  
    "optimizer": opt,  
    "loss_func": criterion,  
    "train_dl": train_dl,  
    "val_dl": val_dl,  
    "sanity_check": False,  
    "lr_scheduler": lr_scheduler,  
    "path2weights": path2models+"weights.pt",}  
model,loss_hist,_=train_val(model,params_train)
```

The training process will start and print the following output:

```
Epoch 0/99, current lr=1e-06  
Copied best model weights!  
train loss: 88394.663371  
val loss: 60473.278360  
-----  
Epoch 1/99, current lr=1e-06  
Copied best model weights!  
train loss: 72682.170562  
val loss: 56371.689775
```

```
-----  
.  
.  
.
```

4. After training is complete, we can plot the progress of the training and validation losses:

```
num_epochs=params_train["num_epochs"]  
  
plt.title("Train-Val Loss")  
plt.plot(range(1,num_epochs+1),loss_hist["train"],label="train")  
plt.plot(range(1,num_epochs+1),loss_hist["val"],label="val")  
plt.ylabel("Loss")  
plt.xlabel("Training Epochs")  
plt.legend()  
plt.show()
```

In the next section, we will describe how each step works.

How it works...

In *step 1*, we defined the `loss_epoch` helper function.

The function inputs are as follows:

- `model`: An object of the model
- `loss_func`: An object of the loss function
- `dataset_dl`: An object of the data loader
- `sanity_check`: A Boolean flag where the default value is `False`
- `opt`: An object of the optimizer

In this function, we extracted data batches from the data loader as `xb` and `yb` tensors. Next, we obtained the model output and calculated the loss value per data batch. We repeated this process for the entire dataset and returned the average loss value per epoch.

In *step 2*, we defined the `train_val` helper function. The function inputs are as follows:

- `model`: An object of the mode
- `params`: A Python dictionary containing the training parameters

In this function, we trained the model for `num_epochs` iterations. In each iteration, we set the model in `train` mode and trained the model for an epoch. Then, we evaluated the model on the validation dataset. We stored the model parameters if the validation results improved in each iteration. Also, we used the learning rate schedule to reduce the learning rate if the validation performance reached a plateau. The function returns the trained model and a dictionary containing the loss values for each iteration.

In *step 3*, we set the training parameters in `params_train` and called the `train_val` function to train the model. You can set the `sanity_check` flag to `True` to quickly execute a few iterations of the training loop and fix any possible errors. You can also change `num_epochs`, depending on the training progress. Typically, if training plateaus quickly, a small value for `num_epochs` is enough since you will not get any improvements by doing any extra training.

In *step 4*, we plotted the results of the training and validation loss values to see our progress in training over multiple epochs.

8

Neural Style Transfer with PyTorch

Who said that deep learning is only for routine tasks? Well, using neural style transfer, you can take everyday images and convert them into artistic masterpieces. The neural style transfer algorithm was first presented in the following article by Gatys et al.: *A Neural Algorithm of Artistic Style* (<https://arxiv.org/abs/1508.06576>).

By way of an example, check out the following screenshots:



The image on the left is converted to the image on the right using a style image by Erin Loree available at the following link: <http://www.erinloreel.com/2012>.

In neural style transfer, we take a content image and a style image. Then, we generate an image to have the content of the content image and the artistic style of the style image. The algorithm is very intuitive to understand and follows the same concepts that we have been learning throughout this book, but with an added twist. By using the masterpieces of great artists as the style image, you can generate very interesting images, as we will see in this chapter.

In this chapter, you will learn to implement the neural style transfer algorithm using PyTorch in order to generate artistic-looking images. You will learn about content loss, style loss, Gram matrix, and how you can use them to implement the algorithm.

This chapter includes the following recipes:

- Loading the data
- Implementing neural style transfer

Loading the data

As mentioned previously, in a neural style transfer algorithm, we require at least a content image and a style image. In this recipe, we will first load the content and style images as PIL objects. Then, we will transform the PIL objects to PyTorch tensors so that we can feed them into the model. Then, we will display the tensors.

Getting ready

For the steps in this chapter, you will need a content image and a style image. You can use the images provided with the scripts of this chapter or use any other images of your own. If you choose the latter, rename your images to `content.jpg` and `style.jpg` and put them into a folder named `data` in the same location of the scripts used in this chapter. The content image can be an everyday picture. For the style image, try to find an artistic image like the ones on the following web page: <https://www.wikiart.org/>.

How to do it...

First, we will prepare the content and style images:

1. Load the content and style images:

```
from PIL import Image
path2content= "./data/content.jpg"
path2style= "./data/style.jpg"
content_img = Image.open(path2content)
style_img = Image.open(path2style)
```

Display the content image:

```
content_img
```

The preceding snippet will display the following screenshot:



The preceding screen-shot shows the image of the slender-horned gazelle at the Cincinnati Zoo, obtained from its wikipedia page, https://en.wikipedia.org/wiki/Rhim_gazelle.

Display the style image:

```
style_img
```

The preceding snippet will display style image. For copyright reasons, we will not show the style image here. Please see **Around and Around** art work by Erin Loree in the following link:

<http://www.erinloreel.com/2012>.

2. Define the image transformations as follows:

```
import torchvision.transforms as transforms

h, w = 256, 384
mean_rgb = (0.485, 0.456, 0.406)
std_rgb = (0.229, 0.224, 0.225)

transformer = transforms.Compose([
    transforms.Resize((h,w)),
    transforms.ToTensor(),
    transforms.Normalize(mean_rgb, std_rgb)])
```

3. Transform the PIL images to tensors:

Pass the content image to transformer:

```
content_tensor = transformer(content_img)
print(content_tensor.shape, content_tensor.requires_grad)
```

The preceding code snippet will print the following output:

```
torch.Size([3, 256, 384]) False
```

Pass the style image to transformer:

```
style_tensor = transformer(style_img)
print(style_tensor.shape, style_tensor.requires_grad)
```

The preceding code snippet will print the following output:

```
torch.Size([3, 256, 384]) False
```

4. Display the content image after transformation:

First, define a helper function to convert back tensors to PIL images:

```
import torch

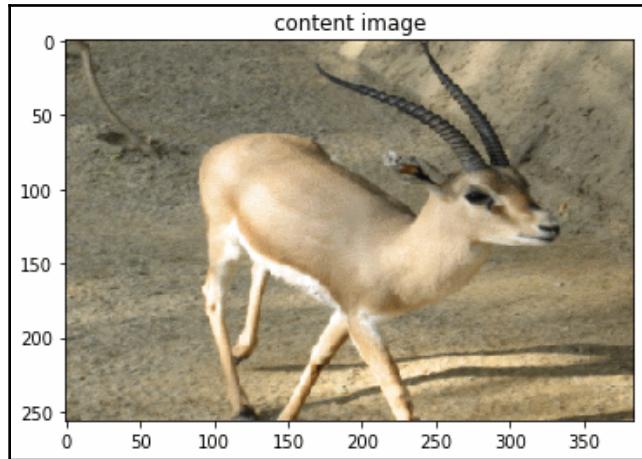
def imgtensor2pil(img_tensor):
    img_tensor_c = img_tensor.clone().detach()
    img_tensor_c*=torch.tensor(std_rgb).view(3, 1,1)
    img_tensor_c+=torch.tensor(mean_rgb).view(3, 1,1)
    img_tensor_c = img_tensor_c.clamp(0,1)
    img_pil=to_pil_image(img_tensor_c)
    return img_pil
```

Then, call the helper function to display content_tensor:

```
import matplotlib.pyplot as plt
%matplotlib inline
from torchvision.transforms.functional import to_pil_image

plt.imshow(imgtensor2pil(content_tensor))
plt.title("content image")
```

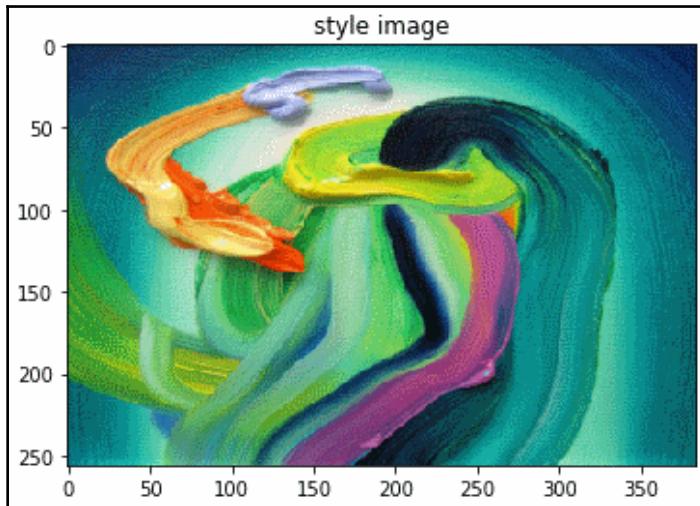
The preceding code snippet will display the following screenshot:



Next, display `style_tensor`:

```
plt.imshow(imgtensor2pil(style_tensor))  
plt.title("style image")
```

The preceding code snippet will display the following screenshot:



As was observed, both content and style images were resized to (256, 384). In the next recipe, we will load the pretrained model.

How it works...

In this recipe, we implemented the code to load the style and content images and transform them into tensors such that we can feed them into the model.

In *step 1*, we loaded the content and style images as `PIL` objects. We used the two images that were provided with the scripts of this chapter. However, you can replace them with your own content and style images. Make sure to copy the images into a folder named `data` in the same location as your scripts.

In *step 2*, we transformed the content and style images into tensors. To this end, we applied three transformations from the `torchvision.transforms` package: `Resize`, `ToTensor`, and `Normalize`.

We used the `Resize` method to resize the images to `(256, 3384)`. Theoretically, you can use images of any size as the content and style images. In practice, these sizes are limited by the amount of memory available on your computer or CUDA device. Also, the higher image size will take a longer time to finish the algorithm.

We then converted `PIL` objects to PyTorch tensors using the `ToTensor` method. This function will scale the pixels' values to the range of `[0, 1]`. Next, we normalized the tensors by subtracting the RGB mean and dividing it by the RGB standard deviation (STD). The predefined RGB mean and standard deviation values depend on the pretrained model that you use for feature extraction. In our case, the VGG19 model was pretrained on the ImageNet dataset and the RGB mean and STD were obtained from the latter.

In *step 3*, we used the transformation function defined in *step 2* to convert the content and style images to normalized tensors. Check out the tensor shape. As you can see, images were resized and reshaped into tensors of shape `[3, 256, 384]`. Also, check out the `requires_grad` attribute of the tensors. Since no optimization is applied to the content and style images, this attribute should be `False`.

In *step 4*, we displayed the content and style tensors. To be able to visualize the tensors, we defined the `imgtensor2pil` helper function to convert the tensors back to `PIL` images. Note that this helper function was defined for visualization purposes only. The input to the helper function is as follows:

- `img_tensor`: A PyTorch tensor of shape `[3, height, width]`

In the function, we first cloned the tensor to prevent making any changes to the original tensor. Since the tensor was previously normalized using the zero-mean-unit-variance normalization method, we renormalized it back to its original values. This was done by multiplying its values by the RGB standard deviation and then adding the RGB mean of the ImageNet dataset. Next, we made sure that the values are in the range $[0, 1]$ by using the `.clamp()` method. Finally, the tensor is converted to a PIL image using the `to_pil_image` function from the `torchvision` package.

Implementing neural style transfer

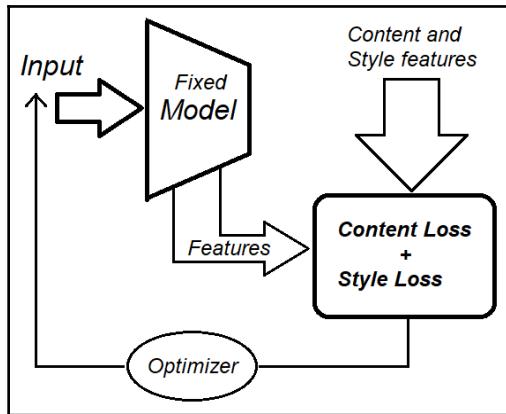
As you have seen throughout this book, we have followed a few standard steps: we loaded the input and target data, defined a model, an objective function and optimizer, and then trained the model by updating the model parameters using the gradient-descent algorithm. In all of these past cases, the input to the model was kept unchanged during the training process and the model was updated.

Now, imagine a situation whereby we keep the model parameters fixed and instead update the input to the model during training. This twist is the intuition behind the neural style transfer algorithm.

Specifically, the neural style transfer algorithm works as follows:

1. Take a pretrained classification model (for example, VGG19), remove the last layers, and keep the remaining layers to serve as a feature extractor.
2. Feed the content image to the model and get selected features to serve as the target content.
3. Feed the style image to the model and get the Gram matrix of selected features to serve as the target style.
4. Feed the input to the model and get the features and the Gram matrix of selected features to serve as the predicted content and style, respectively.
5. Compute the content and style errors, and use this information to update the input and reduce the error.
6. Repeat step 4 until the error is minimized.

A block diagram of the neural style transfer algorithm is shown in the following screenshot:



In the following sections, we will download sample content and style images, and implement the algorithm.

How to do it...

We will develop a neural style transfer algorithm. The algorithm has multiple steps and we will present each major step in a sub-recipe.

Loading the pretrained model

We will load VGG19 as the pretrained model and freeze its parameters:

1. Load the pretrained model:

```
import torchvision.models as models

device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")
model_vgg =
models.vgg19(pretrained=True).features.to(device).eval()

print(model_vgg)
```

The preceding code snippet will print the following output:

```
Sequential(
(0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
```

```
    1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    .
    .
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (35): ReLU(inplace)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
)
```

2. Freeze the model parameters:

```
for param in model_vgg.parameters():
    param.requires_grad_(False)
```

In the next section, we will define the loss functions.

Defining loss functions

We will define helper functions to get the intermediate features from the model and compute the Gram matrix. Then, we will define the content loss and style loss.

1. Define a helper function to get the outputs of intermediate layers:

```
def get_features(x, model, layers):
    features = {}
    for name, layer in enumerate(model.children()):
        x = layer(x)
        if str(name) in layers:
            features[layers[str(name)]] = x
    return features
```

2. Define a helper function to compute the Gram matrix of a tensor:

```
def gram_matrix(x):
    n, c, h, w = x.size()
    x = x.view(n*c, h * w)
    gram = torch.mm(x, x.t())
    return gram
```

3. Define a helper function to compute the content loss:

```
import torch.nn.functional as F

def get_content_loss(pred_features, target_features, layer):
    target = target_features[layer]
    pred = pred_features[layer]
    loss = F.mse_loss(pred, target)
    return loss
```

4. Define a helper function to compute the style loss:

```
def get_style_loss(pred_features, target_features,
style_layers_dict):
    loss = 0
    for layer in style_layers_dict:
        pred_fea = pred_features[layer]
        pred_gram = gram_matrix(pred_fea)
        n, c, h, w = pred_fea.shape
        target_gram = gram_matrix(target_features[layer])
        layer_loss = style_layers_dict[layer] *
        F.mse_loss(pred_gram, target_gram)
        loss += layer_loss/ (n* c * h * w)
    return loss
```

5. Get the features for the content and style images:

```
feature_layers = {'0': 'conv1_1',
                  '5': 'conv2_1',
                  '10': 'conv3_1',
                  '19': 'conv4_1',
                  '21': 'conv4_2',
                  '28': 'conv5_1'}

con_tensor = content_tensor.unsqueeze(0).to(device)
sty_tensor = style_tensor.unsqueeze(0).to(device)

content_features = get_features(con_tensor, model_vgg,
feature_layers)
style_features = get_features(sty_tensor, model_vgg,
feature_layers)
```

For debugging purposes, let's print the shape of the content features:

```
for key in content_features.keys():
    print(content_features[key].shape)
```

The preceding code snippet will print the following output:

```
torch.Size([1, 64, 256, 384])
torch.Size([1, 128, 128, 192])
torch.Size([1, 256, 64, 96])
torch.Size([1, 512, 32, 48])
torch.Size([1, 512, 32, 48])
torch.Size([1, 512, 16, 24])
```

In the next section, we will define the optimizer.

Defining the optimizer

First, we will initialize the input tensor and the optimizer:

1. Initialize the input tensor with the content tensor:

```
input_tensor = con_tensor.clone().requires_grad_(True)
```

2. Define the optimizer:

```
from torch import optim
optimizer = optim.Adam([input_tensor], lr=0.01)
```

In the next section, we will develop and run the algorithm.

Running the algorithm

We will run the neural style transfer algorithm to generate an artistic image:

1. Set the hyperparameters:

```
num_epochs = 301
content_weight = 1e1
style_weight = 1e4
content_layer = "conv5_1"
style_layers_dict = { 'conv1_1': 0.75,
                      'conv2_1': 0.5,
                      'conv3_1': 0.25,
                      'conv4_1': 0.25,
                      'conv5_1': 0.25}
```

2. Run the algorithm:

```
for epoch in range(num_epochs+1):
    optimizer.zero_grad()
    input_features = get_features(input_tensor, model_vgg,
feature_layers)
    content_loss = get_content_loss (input_features,
content_features, content_layer)
    style_loss = get_style_loss(input_features, style_features,
style_layers_dict)
    neural_loss = content_weight * content_loss + style_weight *
style_loss
    neural_loss.backward(retain_graph=True)
    optimizer.step()
    if epoch % 100 == 0:
        print('epoch {}, content loss: {:.2}, style loss
{:.2}'.format(
            epoch, content_loss, style_loss))
```

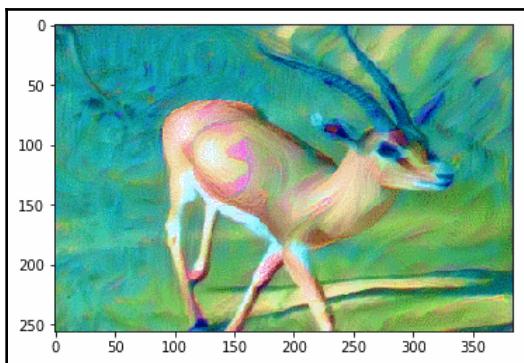
The preceding code snippet, when executed, will print the following output:

```
epoch 0, content loss: 0.0, style loss 3.3e+02
epoch 100, content loss: 3.6, style loss 9e+01
epoch 200, content loss: 3.9, style loss 5.6
epoch 300, content loss: 4.0, style loss 3.3
```

3. Display the result:

```
plt.imshow(imgtensor2pil(input_tensor[0].cpu()));
```

The preceding code snippet will display the following screenshot:



In the next section, we will explain how each step works in detail.

How it works...

In the *Loading the pretrained model* sub-section, we loaded a pretrained model to be used as the feature extractor.

As was observed in *step 1*, we used the `VGG19` model from the `torchvision` package pretrained on the ImageNet dataset. The model expects the input to be mini-batches of shape `(3, Height, Width)` and normalized. This is why we transformed the content and style images in the *Loading the data* section.

In *step 2*, we froze the model parameters using the `requires_grad_` method to avoid any changes to the model during the algorithm optimization.

In the *Defining loss functions* sub-section, we defined the content and style loss functions. To this end, we defined a few helper functions.

In *step 1*, we defined the `get_features` helper function. This helper function was developed to get the intermediate features of the pretrained model. These features will be used in calculating the style and content loss values. The inputs to the helper function are as follows:

- `x`: A PyTorch tensor of shape `[1, 3, height, width]`
- `model`: An object of the pretrained model
- `layers`: A Python dictionary, containing the layer names and numbers

In the helper function, we iterated over the model layers and passed the input tensor `x` to each layer and obtained its output. If the layer name is included in the `layers` dictionary, we collected its output. The helper function returns the collected features as a dictionary.

In *step 2*, we defined the `gram_matrix` helper function to compute the Gram matrix of a tensor. The Gram matrix will be used to calculate the style loss value. The function input is as follows:

- `x`: A tensor of shape `[1, c, h, w]`, where `c`, `h`, `w` are the number of channels, height, and width of `x`

The input tensor `x` comes from the intermediate features of the model. In the helper function, we reshaped `x` from a 4D tensor to a 2D tensor and then calculated the Gram matrix. The output is a tensor of shape `[c, c]`.

In *step 3*, we defined the `get_content_loss` helper function to compute the content loss value. The inputs to the helper function are as follows:

- `pred_features`: A Python dictionary containing the intermediate features of the model given the `input` tensor
- `target_features`: A Python dictionary containing the intermediate features of the model given the `content` tensor
- `layer`: A string containing the layer name

Like any typical loss function, here, we want to compute the distance between the target and predicted values. In the helper function, we extracted the target and predicted tensors for the layer specified by the argument `layer`. Then, we calculated the **mean squared error (MSE)** between the two tensors and returned its value.

In *step 4*, we defined the `get_style_loss` helper function to compute the style loss. The inputs to the helper function are as follows:

- `pred_features`: A Python dictionary containing intermediate features of the model given the `input` tensor
- `target_features`: A Python dictionary containing intermediate features of the model given the `style` tensor
- `style_layers_dict`: A Python dictionary containing the name and weight of the layers included in the style loss

In the helper function, we iterated over the style layers and extracted the predicted and target tensors per layer. Then, we calculated the Gram matrix for the two tensors and used them to compute the MSE. The loss value was calculated per layer, multiplied by the layer weight, normalized and then added all together. The function returned the accumulated loss value for all the layers included in the style loss.

In *step 5*, we called the `get_features` helper function to get the content and style features. To get the features, we passed the content tensor and style tensor to the helper function. Notice that we added a dimension to the tensors using the `.unsqueeze` method since the model input shape is `[1, 3, height, width]`. The name and number of the layers were defined in the Python dictionary, `feature_layers`.

For debugging purposes only, we printed the shape of the content features in each layer.

In the *Defining the optimizer* sub-section, we defined the input tensor and optimizer to be able to update the input based on the loss values.

In *step 1*, we defined the input tensor. If you recall, the goal in the neural style transfer algorithm is to update the input to minimize the loss function. The input can be initialized randomly or with the content image. As was observed, we cloned the content tensor as the input tensor. Notice that the `requires_grad` method should be set to `True` since we want to be able to update the input tensor.

In *step 2*, we defined the optimizer. We chose the `Adam` optimizer from the `torch.optim` package. Notice that we passed `input_tensor` as the parameters to the optimizer. You can also use other optimizers available in the `optim` package, such as `LBFGS`.

In the *Running the algorithm* sub-section, we pieced together all the ingredients and executed the neural style transfer algorithm.

In *step 1*, we defined the hyperparameters. Check out the parameters, `content_weight` and `style_weight`. These parameters define the contributions of the content loss and style loss in the overall loss value. A higher `style_weight` parameter is usually desirable compared to `content_weight`, but you can play around with the values to see their contributions.

The `content_layer` parameter defines the name of the layer included in the content loss. Here, we used `conv5_1` as the content layer. You can set this parameter to a different layer and see the impact on the outcome.

The `style_layer_dict` parameter is a dictionary that defines the name and weight of the layers included in the style loss. As was observed, five layers were included. This parameter can also be changed as you desire. You can remove a layer from the dictionary or change the weights and get slightly different outcomes.

In *step 2*, with all the ingredients ready, we ran the complete neural style transfer algorithm. As was observed, the input tensor was passed to the `get_features` helper function and `input_features` was obtained. Note that `input_features` serves as the features predicted by the model. Then, we called the `get_content_loss` helper function to get `content_loss`. Next, we called the `get_style_loss` helper function to get `style_loss`. Then, the total loss was calculated by combining the content and style losses. Next, using the `.backward` method, the gradients of total loss with respect to the input tensor were calculated. Finally, the input tensor was updated using the `.step` method of the optimizer.

As was observed, the optimizer updated the input tensor as the optimization progressed. The content loss started at zero since we initialized the input tensor with the content tensor and gradually increased. The style loss started at a higher value and gradually decreased as the input tensor was updated.

In *step 3*, we displayed the outcome (`input_tensor`) once optimization was complete. Notice that we used the `imgtensor2pil` helper function to convert the tensor to a `PIL` image. As was observed, our neural style transfer algorithm worked and the resulting image got the texture of the style image.

You can now try using different images as the content or style images and generate new artistic images.

See also

Instead of the `Adam` optimizer, you can use other optimizers such as `LBFGS`, as was proposed in the original paper.

Try `LBFGS` from the `torch.optim` package and see how it can impact the output.

Also, we initialized the input tensor with the content tensor. You can try initializing the input tensor with random values and rerun the algorithm. You may need to play around with the hyperparameters to get your desired outcome.

If you want to try different style images, check out the following links:

- <https://www.wikiart.org/>
- <https://neuralstyle.art/styles>

There are a few online websites and apps that provide style transfer services. Visit the following links:

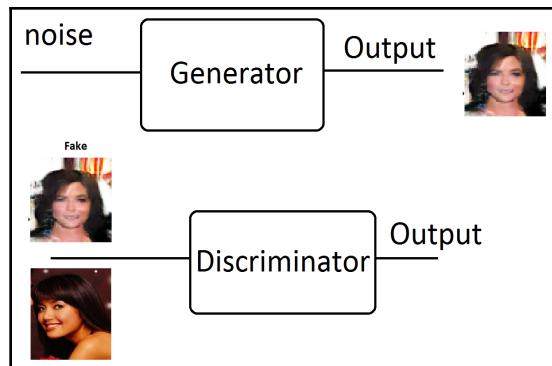
- <https://deepart.io/>
- <https://neuralstyle.art/>

9

GANs and Adversarial Examples

In Chapter 8, *Neural Style Transfer with PyTorch*, we learned one way of generating new data by mimicking the style of artistic images. In this chapter, we will introduce another method of generating new data called **Generative Adversarial Networks (GANs)**. A GAN is a framework that's used to generate new data by learning the distribution of data. It was first introduced by Ian Goodfellow *et al.* in the following landmark article, *Generative Adversarial Nets*, available at <https://arxiv.org/pdf/1406.2661.pdf>.

The GAN framework is comprised of two neural networks, the **generator**, and **discriminator**, as seen in the following diagram:



In the context of image generation, the generator generates fake data, when given noise as input, and the discriminator classifies real images from fake images. During training, the generator and the discriminator compete with each other in a game and as a result, get better at their jobs. The generator tries to generate better-looking images to fool the discriminator and the discriminator tries to get better at identifying real images from fake images.

GANs are still evolving and new applications emerge every day. Some of these applications include artistic image generation, data augmentation, image-to-image translation, super resolutions, and video synthesis.

In this chapter, we will develop a GAN to generate new images similar to the STL-10 dataset, using PyTorch. We will follow the **Deep Convolutional GAN (DCGAN)** architecture presented in the following paper, *Unsupervised representation learning with deep convolutional generative adversarial networks*, available at <https://arxiv.org/abs/1511.06434>.

In addition, in this chapter, we will introduce a vulnerability called adversarial examples or attacks. Adversarial examples are a type of input data that can significantly change a model prediction without being noticeable to the human eye. Due to this fact, adversarial examples can be worrisome, especially in critical tasks such as security or healthcare domains. It would be beneficial to learn how these attacks work in order to start thinking about possible solutions.

This chapter will cover the following recipes:

- Creating the dataset
- Defining the generator and discriminator
- Defining the loss and optimizer
- Training the models
- Deploying the generator
- Attacking models with adversarial examples

Creating the dataset

To train a GAN, we need a training dataset. Given a training dataset, the GAN will learn to generate new data with the same distribution as the training dataset. For instance, if we train a GAN on cat images, it will learn to generate new cat images that look real to our eyes. We will use the STL-10 dataset from the `torchvision` package. We used this dataset in Chapter 3, *Multi-Class Image Classification* for a multi-label classification task. Please see that chapter for more details about the dataset.

In this recipe, you will learn how to define a PyTorch dataset and dataloader to train the GAN framework.

How to do it...

We will create an object of the STL-10 built-in dataset class and define a dataloader as follows:

1. Import the essential packages:

```
from torchvision import datasets
import torchvision.transforms as transforms
import os

path2data = "./data"
os.makedirs(path2data, exist_ok=True)
```

2. Define the data transformations:

```
h, w = 64, 64
mean = (0.5, 0.5, 0.5)
std = (0.5, 0.5, 0.5)
transform = transforms.Compose([
    transforms.Resize((h, w)),
    transforms.CenterCrop((h, w)),
    transforms.ToTensor(),
    transforms.Normalize(mean, std)])
```

3. Instantiate an object of the STL-10 class:

```
train_ds = datasets.STL10(path2data, split='train',
                           download=True,
                           transform=transform)
print(len(train_ds))
```

This snippet will print the following output:

```
5000
```

4. Get a sample tensor from the dataset:

```
import torch
for x, _ in train_ds:
    print(x.shape, torch.min(x), torch.max(x))
    break
```

This snippet will print the following output:

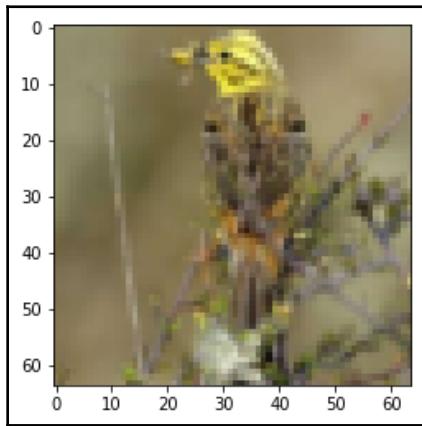
```
torch.Size([3, 64, 64]) tensor(-0.8980) tensor(0.9529)
```

5. Let's display the sample tensor:

```
from torchvision.transforms.functional import to_pil_image
import matplotlib.pyplot as plt
%matplotlib inline

plt.imshow(to_pil_image(0.5*x+0.5))
```

This snippet will display the following screenshot:



6. Create the dataloader:

```
import torch

batch_size = 32
train_dl = torch.utils.data.DataLoader( celeb_ds,
                                         batch_size=batch_size,
                                         shuffle=True)
```

7. Get a data batch from the dataloader:

```
for x, y in train_dl:
    print(x.shape, y.shape)
    break
```

This snippet will print the following output:

```
torch.Size([32, 3, 64, 64]) torch.Size([32])
```

In the next section, we will explain each step in detail.

How it works...

In *step 1*, we imported the essential packages and defined and created a folder to store the data upon downloading it.

In *step 2*, we defined the data transformations using the `torchvision.transforms` package. The original images might be in different sizes, thus we used a `Resize` transformation to resize images to 64 by 64. Next, `ToTensor` scales the image pixels to the range of [0, 1]. Next, we applied a normalization. The normalization `mean` and `std` values were set to normalize inputs to the range of [-1, 1]. As you will find out in *Defining the Generator and Discriminator* recipe, the output of the generator model is a `tanh` function that generates outputs in the range [-1, 1].



The output of the `sigmoid` activation function is in the range [0, 1], while the output of the `tanh` activation function is in the range [-1, +1].

In *step 3*, we instantiated an object of the `STL-10` class from the `torchvision.datasets` package. We passed the location of the data and transformation function to the class. As seen, the dataset has 5000 data samples.

In *step 4*, we got a sample image from the dataset and printed its shape and minimum and maximum values. As expected, the extracted sample is a PyTorch tensor in the shape of (3, height, width) and is normalized to the range of [-1, 1].

In *step 5*, we displayed the sample image. Notice that since the tensor was normalized to [-1, 1], we had to re-normalize it for visualization purposes, as otherwise, the image would look saturated.

In *step 6*, we defined a dataloader. The batch size was set to 32. However, you can adjust it based on your computer and GPU memory.



If you run into memory errors when training a model, try reducing the batch size.

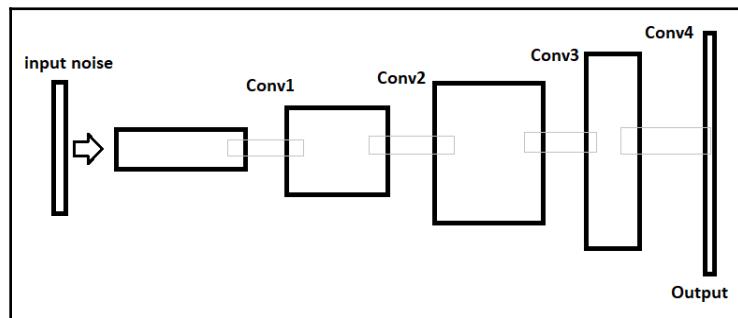
In *step 7*, we extracted a sample mini-batch from the dataloader and printed its shape.

Defining the generator and discriminator

The GAN framework is based on the competition of two models, namely, the generator and discriminator. The generator generates fake images and the discriminator tries to identify real images from fakes images. As a result of this competition, the generator will generate better-looking fake images while the discriminator will become better at identifying them.

As mentioned at the beginning of this chapter, we will follow the DCGAN framework. Based on DCGAN, the architecture of the discriminator is similar to binary classification models based on convolutional layers. We developed a binary classification model in [Chapter 2, Binary Image Classification](#), although here, pooling layers are replaced with stridden convolution layers.

Also, the architecture of the generator is based on the convolutional-transpose layer to upsample the input noise vector to the desired output size, as seen in the following diagram:



In this recipe, you will learn how to implement the generator and discriminator models of our GAN framework.

How to do it...

We will define the generator and discriminator models and initialize their weights:

1. Define the Generator class:

```
from torch import nn
import torch.nn.functional as F

class Generator(nn.Module):
    def __init__(self, params):
        super(Generator, self).__init__()
```

```

        nz = params["nz"]
        ngf = params["ngf"]
        noc = params["noc"]
        self.dconv1 = nn.ConvTranspose2d( nz, ngf * 8,
                                         kernel_size=4, stride=1,
                                         padding=0, bias=False)
        self.bn1 = nn.BatchNorm2d(ngf * 8)
    
```

The Generator class continues with the following code:

```

        self.dconv2 = nn.ConvTranspose2d(ngf * 8, ngf * 4,
                                       kernel_size=4, stride=2,
                                       padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(ngf * 4)
        self.dconv3 = nn.ConvTranspose2d( ngf * 4, ngf * 2,
                                       kernel_size=4, stride=2,
                                       padding=1, bias=False)
        self.bn3 = nn.BatchNorm2d(ngf * 2)
        self.dconv4 = nn.ConvTranspose2d(ngf * 2, ngf,
                                       kernel_size=4, stride=2,
                                       padding=1, bias=False)
        self.bn4 = nn.BatchNorm2d(ngf)
        self.dconv5 = nn.ConvTranspose2d(ngf, noc, kernel_size=4,
                                       stride=2, padding=1,
                                       bias=False)
    
```

The Generator class continues with the following code:

```

def forward(self, x):
    x = F.relu(self.bn1(self.dconv1(x)))
    x = F.relu(self.bn2(self.dconv2(x)))
    x = F.relu(self.bn3(self.dconv3(x)))
    x = F.relu(self.bn4(self.dconv4(x)))
    out = torch.tanh(self.dconv5(x))
    return out
    
```

2. Define an object of the Generator class:

```

params_gen = {
    "nz": 100,
    "ngf": 64,
    "noc": 3,
}
model_gen = Generator(params_gen)
device = torch.device("cuda")
model_gen.to(device)
print(model_gen)
    
```

This snippet will print the following output:

```
Generator(
    (dconv1): ConvTranspose2d(100, 512, kernel_size=(4, 4),
    stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
    track_running_stats=True)
    (dconv2): ConvTranspose2d(512, 256, kernel_size=(4, 4),
    stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
    track_running_stats=True)
    ...
)
```

Let's pass some dummy input to the model:

```
with torch.no_grad():
    y = model_gen(torch.zeros(1, 100, 1, 1, device=device))
    print(y.shape)
```

The preceding snippet will print the following output:

```
torch.Size([1, 3, 64, 64])
```

3. Define the Discriminator class:

```
class Discriminator(nn.Module):
    def __init__(self, params):
        super(Discriminator, self).__init__()
        nic = params["nic"]
        ndf = params["ndf"]
        self.conv1 = nn.Conv2d(nic, ndf, kernel_size=4, stride=2,
        padding=1, bias=False)
        self.conv2 = nn.Conv2d(ndf, ndf * 2, kernel_size=4,
        stride=2, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(ndf * 2)
        self.conv3 = nn.Conv2d(ndf * 2, ndf * 4, kernel_size=4,
        stride=2, padding=1, bias=False)
        self.bn3 = nn.BatchNorm2d(ndf * 4)
        self.conv4 = nn.Conv2d(ndf * 4, ndf * 8, kernel_size=4,
        stride=2, padding=1, bias=False)
        self.bn4 = nn.BatchNorm2d(ndf * 8)
        self.conv5 = nn.Conv2d(ndf * 8, 1, kernel_size=4, stride=1,
        padding=0, bias=False)
```

The Discriminator class continues with the following code:

```
def forward(self, x):
    x = F.leaky_relu(self.conv1(x), 0.2, True)
    x = F.leaky_relu(self.bn2(self.conv2(x)), 0.2, inplace = True)
```

```

x = F.leaky_relu(self.bn3(self.conv3(x)), 0.2, inplace = True)
x = F.leaky_relu(self.bn4(self.conv4(x)), 0.2, inplace = True)
out = torch.sigmoid(self.conv5(x))
return out.view(-1)

```

4. Define an object of the Discriminator class:

```

params_dis = {
    "nic": 3,
    "ndf": 64}
model_dis = Discriminator(params_dis)
model_dis.to(device)
print(model_dis)

```

This snippet will print the following output:

```

Discriminator(
  (conv1): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2),
  padding=(1, 1), bias=False)
  (conv2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2),
  padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
  track_running_stats=True)
  ...

```

Let's pass some dummy input to the model:

```

with torch.no_grad():
    y= model_dis(torch.zeros(1,3,h,w, device=device))
    print(y.shape)

```

The preceding snippet will print the following output:

```
torch.Size([1])
```

5. Define a helper function to initialize the model weights:

```

def initialize_weights(model):
    classname = model.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(model.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(model.weight.data, 1.0, 0.02)
        nn.init.constant_(model.bias.data, 0)

```

6. Initialize the model weights by calling the helper function:

```

model_gen.apply(initialize_weights);
model_dis.apply(initialize_weights);

```

In the next section, we will explain each step in detail.

How it works...

In *step 1*, we defined the `Generator` class with two methods. In the `__init__` method, we defined the layers. The function has one input argument, `params`, which is a Python dictionary with the following keys:

- `nz`: The size of the input noise vector (set to 100)
- `ngf`: A coefficient for the number of convolutional filters in the generator (set to 64)
- `noc`: The number of output channels (set to 3 for RGB images)

As seen, five `conv-transpose` layers were defined. A `conv-transpose` layer is also called a fractionally-strided convolution or a deconvolution. They are used to upsample the input vector to the desired output size.

In the `forward` method, we defined the connections between the layers and got the output. The output of `Generator` is a tensor of shape `(batch_size, 3, height, width)`.

In *step 2*, we defined an object of the `Generator` class called `model_gen`. To make sure that the model was created properly, we passed some dummy input to the generator model. As expected, the model output is a tensor of shape `[1, 3, 64, 64]`.

In *step 3*, we defined the `Discriminator` class. Similarly, in the `__init__` method, we defined the layers and in the `forward` method, we defined the connections between the layers. Notice that we did not use any pooling layers and instead set the `stride` argument to 2 or 4 to downsample the input size. Also, notice that `leaky_relu` activation was used instead of `relu` to reduce overfitting.

In *step 4*, we defined an object of the `Discriminator` class. To make sure that the model was created properly, we passed some dummy input to the discriminator model. This simple test will help fix any possible errors before we move on to the next steps.

In *step 5*, we defined a helper function to initialize the model weights. The input to the function was a PyTorch model. The DCGAN paper suggested initializing the weights using a normal distribution with `mean=0` and `std=0.02`, as we did in the helper function.

In *step 6*, we applied an `initialize_weights` helper function to the generator and discriminator models to initialize their weights.

Defining the loss and optimizer

For the models to learn, we need to define a criterion. The discriminator model is a classification network and we can use the **binary cross-entropy (BCE)** loss function as its criterion. For the generator model to learn, we pass its output to the discriminator model and then evaluate the output of the discriminator model. Thus, the same BCE loss function can be used as a criterion to train the generator model. Also, we will use the Adam optimizer to update the parameters of the discriminator and generator models.

In this recipe, you will learn how to define the loss function and optimizer of a GAN network.

How to do it...

We will define the loss function and optimizer:

1. Define an object of the BCE loss class:

```
loss_func = nn.BCELoss()
```

2. Define the optimizer for the generator:

```
lr = 2e-4  
beta1 = 0.5  
opt_dis = optim.Adam(model_dis.parameters(), lr=lr, betas=(beta1,  
0.999))
```

3. Define the optimizer for the discriminator:

```
opt_gen = optim.Adam(model_gen.parameters(), lr=lr, betas=(beta1,  
0.999))
```

In the next section, we will explain how each step works.

How it works...

In *step 1*, we defined the BCE criterion from the `torch.nn` package to calculate the BCE loss between the target and the output. As you will see in the next section, we will use this loss function in multiple steps.

In *step 2*, we defined the Adam optimizer from `torch.optim` for the generator model based on the hyperparameters suggested in the DCGAN paper. The paper suggested setting the learning rate to 0.0002 and the momentum term `beta1` to 0.5 for training stability.

Similarly, in *step 3*, we defined the Adam optimizer from `torch.optim` for the discriminator model.

Training the models

Training the GAN framework is done in two stages: training the discriminator and training the generator. To this end, we will take the following steps:

1. Get a batch of real images with the target labels set to 1.
2. Generate a batch of fake images using the generator with the target labels set to 0.
3. Feed the mini-batches to the discriminator and compute the loss and gradients.
4. Update the discriminator parameters using the gradients.
5. Generate a batch of fake images using the generator with the target labels set to 1.
6. Feed the fake mini-batch to the discriminator and compute the loss and gradients.
7. Update the generator only based on gradients.
8. Repeat from *step 1*.

In this recipe, you will learn how to implement these steps.

How to do it...

We will implement the training steps of our GAN network:

1. Define a few parameters:

```
real_label = 1
fake_label = 0
nz = params["nz"]
num_epochs = 100

loss_history={"gen": [],
              "dis": []}
```

2. Start the training loop and calculate the loss for real samples:

```
batch_count = 0
for epoch in range(num_epochs):
    for xb, yb in celeb_dl:
        ba_si = xb.size(0)
        model_dis.zero_grad()
        xb = xb.to(device)
        yb = torch.full((ba_si,), real_label, device=device)
        out_dis = model_dis(xb)
        loss_r = loss_func(out_dis, yb)
        loss_r.backward()
```

Continue the training loop by calculating the loss for fake samples:

```
noise = torch.randn(ba_si, nz, 1, 1, device=device)
out_gen = model_gen(noise)
out_dis = model_dis(out_gen.detach())
yb.fill_(fake_label)
loss_f = loss_func(out_dis, yb)
loss_f.backward()
loss_dis = loss_r + loss_f
opt_dis.step()
```

Continue the training loop:

```
model_gen.zero_grad()
yb.fill_(real_label)
out_dis = model_dis(out_gen)
loss_gen = loss_func(out_dis, yb)
loss_gen.backward()
opt_gen.step()

loss_history["gen"].append(loss_gen.item())
loss_history["dis"].append(loss_dis.item())
batch_count += 1
if batch_count % 100 == 0:
    print(epoch, loss_gen.item(), loss_dis.item())
```

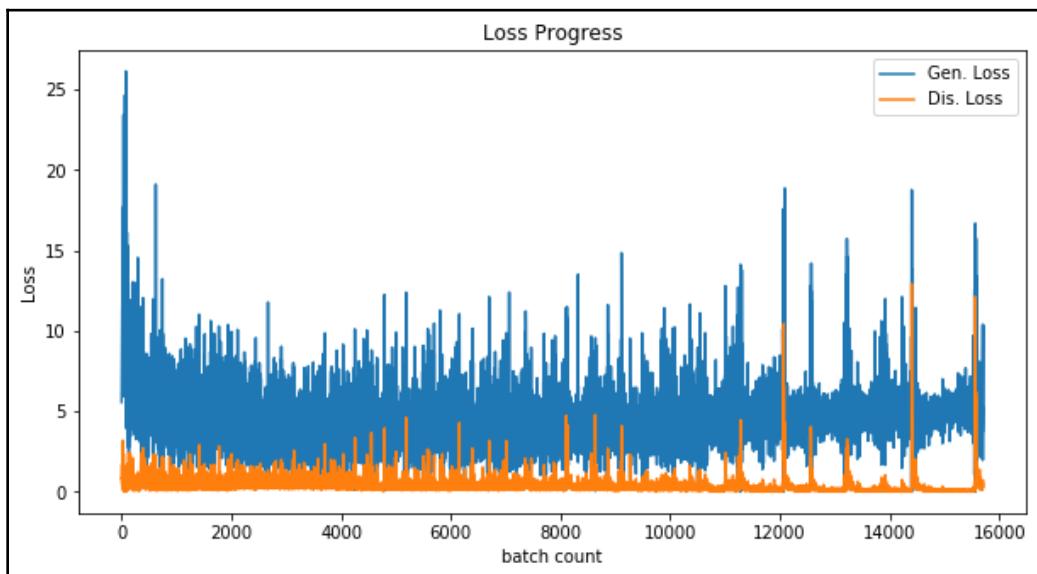
The training will start and will print the following output:

```
0 7.026479721069336 0.12888161838054657
1 3.8994224071502686 0.24403591454029083
1 12.108219146728516 1.221606731414795
...
...
```

3. Plot the loss history:

```
plt.figure(figsize=(10,5))
plt.title("Loss Progress")
plt.plot(loss_history["gen"],label="Gen. Loss")
plt.plot(loss_history["dis"],label="Dis. Loss")
plt.xlabel("batch count")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

This snippet will display the following screenshot:



4. Store the model weights:

```
import os

path2models = "./models/"
os.makedirs(path2models, exist_ok=True)
path2weights_gen = os.path.join(path2models, "weights_gen.pt")
path2weights_dis = os.path.join(path2models, "weights_dis.pt")

torch.save(model_gen.state_dict(), path2weights_gen)
torch.save(model_dis.state_dict(), path2weights_dis)
```

In the next section, we will explain each step in detail.

How it works...

In *step 1*, we defined a few parameters. We defined `real_label` and `fake_label` and set them to 1 and 0, respectively. Later, we will need to label a mini-batch using these parameters. The `nz` parameter specifies the size of the input noise vector to the generator model. This was set to 100 in *Defining the generator and discriminator* recipe.

The `num_epochs` parameter specifies how many times we want to iterate over the training data. To store the loss values for the discriminator and generator models, we defined the `loss_history` dictionary.

In *step 2*, we implemented the training loop. The training loop iterates over the real dataset for `num_epochs`. In each epoch, we got a batch of real images from `celeb_dl` and fed it to the discriminator model and got its output as `out_dis`. Note that here, the real images were labeled with `real_label` using the `torch.full` method. Then, the loss value for the real mini-batch was calculated as `loss_r`. Next, the gradients of `loss_r` with respect to the discriminator parameters were calculated in a backward pass.

Next, we generated a mini-batch of fake images using the generator and fed them to the discriminator. In passing the output of the generator to the discriminator, we used the `.detach()` method to avoid gradient tracking for the generator model. Note that at this point, the fake images were labeled with `fake_label` using the `torch.fill_` method. Then, the loss value for the fake mini-batch was calculated as `loss_f`. Next, the gradients of `loss_f` with respect to the discriminator parameters were calculated in a backward pass.

Finally, we updated the discriminator parameters using the `.opt_dis.step()` method.

Next, we trained the generator model. To this end, we passed the fake images to the discriminator model and got its output. Note that here, the fakes images were labeled with `real_label` using the `.fill_` method. This may sound strange at first, but it is done to force the generator model to generate better-looking images.



Even though the outputs of the generator model are fake, we used `real_label` as the target value when computing the loss value.

Then, we calculated the loss value as `loss_gen`, computed its gradients, and updated the generator parameters using `opt_gen.step()`. By executing the code, the loss values were printed on the screen.

In *step 3*, we plotted the generator and discriminator loss values during training.

In *step 4*, we stored the trained weights into pickle files for future use.

See also

There has been significant progress on GANs in recent years to increase the quality and dimension of generated data. As an example, you can refer to the following paper:

- *StyleGAN: A Style-Based Generator Architecture for Generative Adversarial Networks*, available from <https://arxiv.org/abs/1812.04948>

This paper is from NVIDIA and combined progressive GANs and neural style transfer to generate high-quality images.

Check out a PyTorch implementation of the paper at <https://github.com/roslinality/style-based-gan-pytorch>.

Deploying the generator

Once we've trained a GAN, we end up with two trained models. Usually, we discard the discriminator model and keep the generator model. We can use the trained generator to generate new images. To deploy the generator model, we load the trained weights into the model and then feed it with random noise. Make sure to define the model class beforehand. To avoid repetition, we will not define the model class here.

In this recipe, you will learn how to deploy the generator model.

How to do it...

We will load the weights into the generator model and generate new images:

1. Load the weights:

```
weights = torch.load(path2weights_gen)
model_gen.load_state_dict(weights)
```

2. Set the model in evaluation mode:

```
model_gen.eval()
```

3. Feed noise to the model and get its outputs:

```
with torch.no_grad():
    fixed_noise = torch.randn(16, nz, 1, 1, device=device)
    img_fake = model_gen(noise).detach().cpu()
    print(img_fake.shape)
```

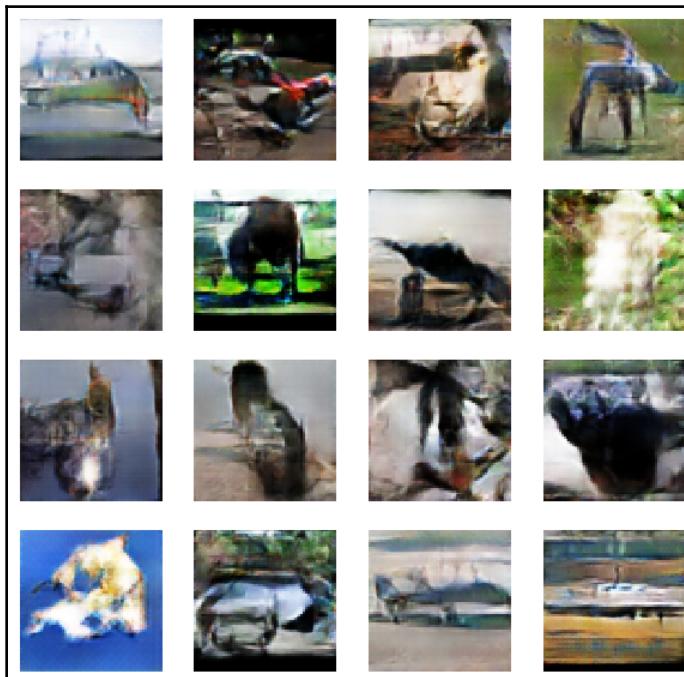
This snippet will print the following output:

```
torch.Size([16, 3, 64, 64])
```

4. Display the generated images:

```
plt.figure(figsize=(10,10))
for ii in range(16):
    plt.subplot(4,4,ii+1)
    plt.imshow(to_pil_image(0.5*img_fake[ii]+0.5))
    plt.axis("off")
```

This snippet will display the following screenshot:



In the next section, we will explain how each step works.

How it works...

In *step 1*, we loaded the trained weights from the pickle file into the generator model. In *step 2*, we set the model in evaluation mode. In *step 3*, we fed random noise vectors into the model and received generated fake images. In *step 4*, we displayed the generated faked images.



Note that we had to re-normalize the output tensor back to its original values for visualization purposes.

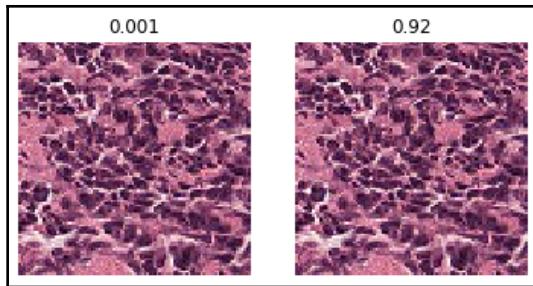
Check out the generated images. Some of them may look very distorted, while others look relatively realistic. To improve the results, you can train the model on a single class of data as opposed to multiple classes together. GANs perform better when they are trained with a single class. The STL-10 dataset has multiple classes. Try to select one category and the train the GAN models. Also, you can try to train the model for a longer time and see how that changes the generated images.

Attacking models with adversarial examples

In the previous chapters, such as Chapters 2 to 6, we have seen the power of deep learning models to solve various computer vision tasks. We trained and then tested multiple models on different datasets. Now, we are going to turn our attention to the robustness of these models.

In this recipe, we want to introduce a vulnerability called adversarial examples or attacks. Adversarial examples are a type of input data that can significantly change the model prediction without being noticeable to the human eye. Due to this fact, adversarial examples can be worrisome, especially in critical tasks such as the security or healthcare domains. It would be beneficial to learn how these attacks work in order to start thinking about possible solutions.

An example of an adversarial attack is shown in the following screenshot:



The image on the left is an original image, while the image on the right is an adversarial version of the same image. Even though both images look the same, the predicted probabilities by a classifier, shown on the titles, are significantly different. In this chapter, we will develop such an adversarial attack.

There are two types of adversarial attacks: white-box and black-box attacks. In white-box attacks, the attacker has the knowledge of the model, input, and loss function that was used to train the model. By using this knowledge, the attacker can change the inputs to disrupt the predicted outputs. The amount of change in the input is usually minor and indistinguishable to the human eye. A common type of white-box attack is called the **Fast Gradient Sign (FGS)** attack, which works by changing the input to maximize the loss.

In the FGS attack, given an input and a pre-trained model, we compute the gradients of the loss with respect to the input. Then, we add a small portion of the absolute value of gradients to the input.

In this recipe, you will learn how to develop an FGS attack for the binary classification model presented in Chapter 2, *Binary Image Classification*, of this book. As you may recall, we developed a classifier on the *Histo dataset* to classify cancerous images. We will use the same dataset and trained model here for our purposes. See [Chapter 2, Binary Image Classification](#), for all the details.

Getting ready

Make sure that you follow the instructions of [Chapter 2, Binary Image Classification](#), of this book to download the data and define the model. You can copy the scripts used in that chapter to build the custom PyTorch dataset and model. For your convenience, the dataset class and model definitions are provided to you in separate Python files called `mydataset.py` and `mymodel.py`, together with the scripts of this chapter. Moreover, the pre-trained weights for the model are provided in the `cnn_weights.pt` file. Please copy these files to the location of your code for future use.

How to do it...

We will load the dataset, load the pre-trained model, and implement the FGS attack.

Loading the dataset

We will import the custom dataset class and define an object of the dataloader:

1. Import the Python file provided to you to build the dataset:

```
import mydataset
```

2. Define an object of the dataloader and get a mini-batch:

```
test_dl = mydataset.test_dl
for xb,yb in test_dl:
    print(xb.shape, yb.shape)
    break
```

The preceding snippet will print the following output:

```
torch.Size([1, 3, 96, 96]) torch.Size([1])
```

In the next section, we will load the pre-trained model.

Loading the pre-trained model

We will load the pre-trained model, freeze its parameters, and verify its performance on the test dataset:

1. Import the pre-trained model:

```
import mymodel
```

2. Define an object of the pre-trained model:

```
model = mymodel.model
```

3. Move the model to the CUDA device:

```
import torch
device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")
model = model.to(device)
```

4. Define a helper function to freeze the model parameters:

```
def freeze_model(model):  
    for child in model.children():  
        for param in child.parameters():  
            param.requires_grad = False  
    print("model frozen")  
    return model
```

5. Call the helper function to freeze the model:

```
model = freeze_model(model)
```

6. Deploy the pre-trained model on some sample test data:

```
from sklearn.metrics import accuracy_score  
  
def deploy_model(model, test_dl):  
    y_pred = []  
    y_gt = []  
    with torch.no_grad():  
        for x,y in test_dl:  
            y_gt.append(y.item())  
            out = model(x.to(device)).cpu().numpy()  
            out = np.argmax(out, axis=1)[0]  
            y_pred.append(out)  
    return y_pred, y_gt  
y_pred, y_gt = deploy_model(model,test_dl)
```

7. Verify the pre-trained model's performance on the sample test data:

```
from sklearn.metrics import accuracy_score  
acc=accuracy_score(y_pred,y_gt)  
print("accuracy: %.2f" %acc)
```

The preceding snippet will print the following output:

```
accuracy: 0.94
```

In the next section, we will implement the adversarial attack.

Implementing the attack

We will implement the attack and display sample adversarial examples:

1. Import the required packages:

```
from torchvision.transforms.functional import to_pil_image
import torch.nn.functional as F
import matplotlib.pyplot as plt
```

2. Get mini-batches from the dataloader and call `perturb_input`:

```
y_pred = []
y_pred_p = []
for xb,yb in test_dl:
    xb_p, out = perturb_input(xb, yb, model, alfa = 0.005)
```

3. Here, the `perturb_input` helper function is defined as follows:

```
def perturb_input(xb, yb, model, alfa):
    xb = xb.to(device)
    xb.requires_grad = True
    out = model(xb).cpu()
    loss = F.nll_loss(out, yb)
    model.zero_grad()
    loss.backward()
    xb_grad = xb.grad.data
    xb_p = xb + alfa * xb_grad.sign()
    xb_p = torch.clamp(xb_p, 0, 1)
    return xb_p, out.detach()
```

4. Calculate the prediction probabilities before and after perturbation:

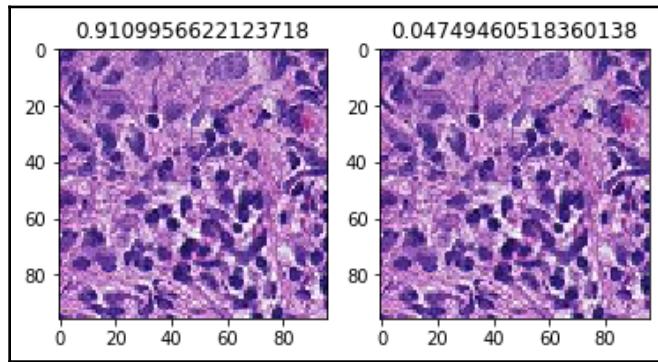
```
with torch.no_grad():
    pred = out.argmax(dim=1, keepdim=False).item()
    y_pred.append(pred)
    prob = torch.exp(out[:, 1])[0].item()

    out_p = model(xb_p).cpu()
    pred_p = out_p.argmax(dim=1, keepdim=False).item()
    y_pred_p.append(pred_p)
    prob_p = torch.exp(out_p[:, 1])[0].item()
```

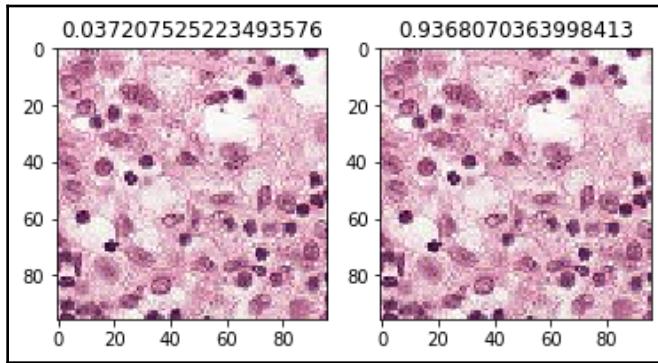
5. Display the original and perturbed input:

```
plt.subplot(1, 2, 1)
plt.imshow(to_pil_image(xb[0].detach().cpu()))
plt.title(prob)
plt.subplot(1, 2, 2)
plt.imshow(to_pil_image(xb_p[0].detach().cpu()))
plt.title(prob_p)
plt.show()
```

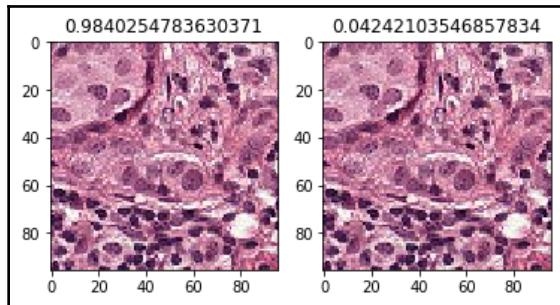
The preceding snippet will display all test images before and after perturbation. A few images are shown in the following screenshot:



Some more can be seen in the following screenshot:



Some more can be seen in the following screenshot:



The images on the left are original, while those on the right are perturbed. Prediction probabilities are shown as the title of each image.

6. Let's compute the model accuracy for the perturbed data:

```
acc=accuracy_score(y_pred_p,y_gt)  
print("accuracy: %.2f" %acc)
```

The preceding snippet will print the following output:

```
accuracy: 0.31
```

In the next section, we will explain each step in detail.

How it works...

In the *Loading the Dataset* subsection, we loaded the dataset using the scripts we developed in Chapter 2, *Binary Image Classification*. In step 1, we imported the `mydataset.py` file. This file contains the custom dataset class. See Chapter 2, *Binary Image Classification*, for all the details of how the dataset class was created. In step 2, we defined an object of the dataloader and extracted a mini-batch. The batch size was set to 1 and as a result, a tensor of shape `(1, 3, 96, 96)` was extracted.

In the *Loading the pre-trained model* subsection, we loaded the pre-trained model that we developed in Chapter 2, *Binary Image Classification*. In step 1, we imported the `mymodel.py` file as a package. This file contains the model class definition. In step 2, we created an object of the model class. In step 3, we moved the model to the CUDA device, if one is available. In step 4, we defined a helper function to freeze the model parameters. This is to avoid any update to the model parameters. Remember that in adversarial attacks, we try to update the input to change the model prediction.

The helper function has one input: `model`. In the function, we iterated over the model parameters and set the `requires_grad` attribute to `False`. In *step 5*, we called the `freeze_model` helper function. In *step 6*, we deployed the pre-trained model on a subset of data called the test dataset. This step is carried out to verify the pre-trained model. In *step 7*, we verified the performance of the pre-trained model on the test dataset. As seen, the current accuracy on the original test dataset is high. This verifies that the pre-trained model was correctly loaded. We will see how this performance can be affected when we introduce adversarial attacks.

In the *Implementing the attack* subsection, we developed the FSG attack. In *step 1*, we imported the required packages. In *step 2*, we started a loop to iterate over the test dataset. We then got a mini-batch from the dataloader and passed it to the `perturb_input` helper function, which is defined in the next step.

In *step 3*, we defined the `perturb_input` helper function. The function inputs are as follows:

- `xb`: The input to be perturbed, a PyTorch tensor of shape `[1, 3, height, width]`
- `yb`: The target label, a tensor of shape `[1]`
- `model`: The pre-trained model
- `alfa`: The perturbation coefficient, set to `0.005`

In the helper function, we first set the `requires_grad` attribute of the input tensor to `True`. This was done to be able to compute the gradients of the loss with respect to the input. Next, we passed the input to the pre-trained model and got its output.

Next, we computed the loss value by comparing the model output and the target label. Then, we computed the gradients of loss with respect to the input tensor. Finally, we perturbed the input by adding a portion of the signed gradients. The `alfa` coefficient defines the amount of distortion added to the input. Higher `alfa` clearly leads to more distortion. We set `alfa=0.005` so that it is small enough to change the predictions while still being invisible to our eyes.

In *step 4*, we fed the perturbed input to the model and got the model prediction. Notice that we stopped tracking gradients at this step using the `torch.no_grad()` method. In *step 5*, we displayed the original input and the perturbed input together with the predicted probabilities. As seen, the amount of distortion is not noticeable in the input while the predictions are almost flipped. In *step 6*, we computed the model accuracy on the perturbed data. As seen, the model accuracy significantly dropped for the distorted data.

There's more...

We used a small coefficient to perturb the input. Try to change the `alfa` coefficient and see its impact on both input images and the model predictions. You can also try to use the FGS attack to fool other models.

Now that you've learned how to attack a model, you may want to think about defending against such attacks!

10

Video Processing with PyTorch

So far, we've only dealt with images. We've built various image classification, detection, and segmentation models. We've even generated new images from practically nothing (noise). But images are still and static. There is no motion in static images. The real joy comes from motion. And that is how videos come into play. But what is a video, anyway?

The truth is that videos are not that more complicated than images. A video is, in fact, a collection of sequential frames or images that are played one after another. This sequence can be seen in the following screenshot:



To get a smooth video, we need to play a certain number of frames per second; otherwise, the video will look disjointed. Most of the videos that we deal with in our daily life have more than 30 frames per second. So, now, you get the idea. With that scale, a short video that's 10 seconds long would be equivalent to 300 images. And this multitude of images makes things complicated.

Despite this complexity, there are many applications that we can use for video processing. Some of these applications are closely related to the image processing applications that we've already studied in this book. For instance, in Chapter 3, *Multi-Class Image Classification*, we developed a multi-class classification model to classify images into different categories. Now that we're dealing with videos, we may be interested in classifying videos too. Such an application is useful if you want to know what kind of activity is happening in a video, as opposed to what objects are present in an image. In this chapter, we will build a video classification model using PyTorch.

In this chapter, we will cover the following recipes:

- Creating the dataset
- Defining the model
- Training the model
- Deploying the video classification model

Creating the dataset

As always, the first step is to create the dataset. We will need a training dataset to train our model and a test or validation dataset to evaluate the model. For this purpose, we will use **HMDB: a large human motion database**, which is available at <https://serre-lab.clps.brown.edu/resource/hmdb-a-large-human-motion-database/#overview>.

The HMDB dataset was collected from various sources, including movies, the Prelinger archive, YouTube, and Google videos. It is a pretty large dataset (2 GB) with a total of 7,000 video clips. There are 51 action classes, each containing a minimum of 101 clips.

An illustration of a few sample actions can be seen in the following image:



In the preceding image, we have only displayed one frame from each video.



For more details about the database, please refer to the following article: H. Kuehne, H. Jhuang, E. Garrote, T. Poggio, and T. Serre. *HMDB: A Large Video Database for Human Motion Recognition*. ICCV, 2011.

To create a dataset for video classification, we will convert the videos into images. Each video has hundreds of frames or images. It is not computationally feasible to process all the frames of a video. To simplify this, we will select 16 frames per video that are equally spaced across the video. Then, we will define a PyTorch dataset class. Next, we will define the PyTorch data loaders for two types of deep learning models: a **recurrent neural network (RNN)** model and a **three-dimensional convolutional neural network (3D-CNN)** model. For more details on the models, see the *Creating the model* recipe.

In this recipe, you will download the HMDB dataset, convert the video clips into images, and define the PyTorch dataset and data loader classes for video classification.

Getting ready

In this section, we will download the HMDB dataset. To download the database, do the following:

1. Visit the following link: <https://serre-lab.clps.brown.edu/resource/hmdb-a-large-human-motion-database/#overview>.
2. Click on the **Download** menu.
3. Click on **HMDB51** to download the dataset.

The downloaded file (`hmdb51_org.rar`) is a compressed file in `.rar` format. On Linux machines, you may need to install the `unrar` program on your computer to be able to extract the videos.



Windows users can use 7-Zip to extract RAR files.

To do so, execute the following command in a Terminal:

```
$ sudo apt-get install unrar
```

Next, extract the `hmdb51_org.rar` file into the `hmdb51_org` folder. The folder should contain 51 subfolders corresponding to 51 class actions. Also, each subfolder should contain at least 101 video files of the `.avi` type for each action class. Try playing a random video clip from a folder to get familiar with the videos.

Then, create a folder named `data` in the same location as your scripts and copy the `hmdb51_org` folder into the `data` folder.

In the next section, we will convert the videos into images and then define the dataset and data loader classes.

How to do it...

We will prepare the data by converting the videos into images. Then, we will define the PyTorch dataset and the `data-loader` classes.

Preparing the data

Let's read some videos and convert them into images. Each video may contain hundreds of images. To simplify this problem, we will only use 16 frames that are equally spaced across the entire video and store them as `.jpg` files:

1. Get a list of categories in the dataset:

```
import os

path2data = "./data"
sub_folder = "hmdb51_org"
sub_folder_jpg = "hmdb51.jpg"
path2aCatgs = os.path.join(path2data, sub_folder)

listOfCategories = os.listdir(path2aCatgs)
len(listOfCategories)
```

The preceding snippet will print the following output:

```
...
'talk',
'throw',
'turn',
'walk',
'wave'],
51)
```

We've only displayed an excerpt of the output for brevity.

2. Get the number of subfolders per action class:

```
for cat in listOfCategories:  
    print("category:", cat)  
    path2acat = os.path.join(path2aCatgs, cat)  
    listOfSubs = os.listdir(path2acat)  
    print("number of sub-folders:", len(listOfSubs))  
    print("-"*50)
```

The preceding snippet will print the following output:

```
category: jump  
number of sub-folders: 151  
-----  
category: shake_hands  
number of sub-folders: 162  
-----  
...
```

We've only displayed an excerpt of the output for brevity.

3. Define two helper functions to get and store the frames from a video. These helper functions are defined in the `myutils.py` file, which is provided to you as part of this book. So, here, we only need to import `myutils`:

```
import myutils
```



You may need to install the `tqdm` package if you do not have it installed on your computer already. To install the `tqdm` package, use the following command:

```
$ pip install tqdm
```

4. Loop over the videos, get the frames, and store them as jpg files:

```
extension = ".avi"  
n_frames = 16  
for root, dirs, files in os.walk(path2aCatgs, topdown=False):  
    for name in files:  
        if extension not in name:  
            continue  
        path2vid = os.path.join(root, name)  
        frames, vlen = myutils.get_frames(path2vid, n_frames=n_frames)  
        path2store = path2vid.replace(sub_folder, sub_folder_jpg)
```

```
path2store = path2store.replace(extension, "")  
print(path2store)  
os.makedirs(path2store, exist_ok= True)  
myutils.store_frames(frames, path2store)  
print("-"*50)
```

The preceding snippet will take some time to finish executing.

In the next section, we will define the dataset class.

Splitting the data

In the previous section, *Preparing the data*, we converted the videos into images. For each video, we extracted 16 frames. Here, we will split the dataset into training and test sets. We provided you with `myutils.py` as part of this book, which contains some of the utility functions required for this recipe:

1. Let's define the data path and import `myutils`:

```
import os  
import myutils  
  
path2data = "./data"  
sub_folder_jpg = "hmdb51_jpg"  
path2ajpgs = os.path.join(path2data, sub_folder_jpg)
```

2. The frames of each video were stored in a folder with the same name as the video. Call the `get_vids` helper function from `myutils` to get a list of video filenames and labels:

```
all_vids, all_labels, catgs = myutils.get_vids(path2ajpgs)  
len(all_vids), len(all_labels), len(catgs)
```

The preceding snippet will print the following output:

```
(6766, 6766, 51)
```

Let's get a snapshot of the outputs that were returned from the `get_vids` helper function:

```
all_vids[:3], all_labels[:3], catgs[:5]
```

The preceding snippet will print the following output:

```
( ['./data/hmdb51_jpg/sword_exercise/samurai_sword_action_lesson_prew_view_sword_exercise_f_cm_np1_le_med_3'],
  ['sword_exercise', 'sword_exercise', 'sword_exercise'],
  ['sword_exercise', 'pushup', 'turn', 'draw_sword', 'clap'])
```

3. Define a Python dictionary to hold the numerical values of the labels:

```
labels_dict = {}
ind = 0
for uc in catgs:
    labels_dict[uc] = ind
    ind+=1
labels_dict
```

The preceding snippet will print the following output:

```
{'sword_exercise': 0,
 'pushup': 1,
 'turn': 2,
 'draw_sword': 3,
 'clap': 4,
 ...}
```

4. As we can see, there is a total of 51 categories. To simplify this problem, we will select five action classes and filter out videos:

```
num_classes = 5
unique_ids = [id_ for id_, label in zip(all_vids, all_labels)
              if
                labels_dict[label]<num_classes]
unique_labels = [label for id_, label in zip(all_vids, all_labels)
                  if
                    labels_dict[label]<num_classes]
len(unique_ids), len(unique_labels)
```

The preceding snippet will print the following output:

```
(703, 703)
```

5. Split the videos into two groups:

```
from sklearn.model_selection import StratifiedShuffleSplit

sss = StratifiedShuffleSplit(n_splits=2, test_size=0.1,
                             random_state=0)
train_indx, test_indx = next(sss.split(unique_ids, unique_labels))
```

```
train_ids = [unique_ids[ind] for ind in train_idx]
train_labels = [unique_labels[ind] for ind in train_idx]
print(len(train_ids), len(train_labels))

test_ids = [unique_ids[ind] for ind in test_idx]
test_labels = [unique_labels[ind] for ind in test_idx]
print(len(test_ids), len(test_labels))
```

The preceding snippet will print the following output:

```
632 632
71 71
```

In the next section, we will define a PyTorch dataset class and instantiate two objects for the training and test datasets.

Defining the PyTorch datasets

In the previous section, *Splitting the data*, we split the data into training and test sets. Here, we will define a PyTorch dataset class. Then, we will instantiate two objects of the class for the training and test datasets:

1. Import the essential packages:

```
from torch.utils.data import Dataset, DataLoader, Subset
import glob
from PIL import Image
import torch
import numpy as np
import random
np.random.seed(2020)
random.seed(2020)
torch.manual_seed(2020)
```

2. Define the dataset class:

```
class VideoDataset(Dataset):
    def __init__(self, ids, labels, transform):
        self.transform = transform
        self.ids = ids
        self.labels = labels
    def __len__(self):
        return len(self.ids)
```

The dataset class continues with the following code:

```
def __getitem__(self, idx):
    path2imgs=glob.glob(self.ids[idx]+"/*.jpg")
    path2imgs = path2imgs[:timesteps]
    label = labels_dict[self.labels[idx]]
    frames = []
    for p2i in path2imgs:
        frame = Image.open(p2i)
        frames.append(frame)

    seed = np.random.randint(1e9)
    frames_tr = []
    for frame in frames:
        random.seed(seed)
        np.random.seed(seed)
        frame = self.transform(frame)
        frames_tr.append(frame)
    if len(frames_tr)>0:
        frames_tr = torch.stack(frames_tr)
    return frames_tr, label
```

3. Define the transformation parameters:

```
# choose one
model_type = "3dcnn"
model_type = "rnn"

timesteps =16
if model_type == "rnn":
    h, w =224, 224
    mean = [0.485, 0.456, 0.406]
    std = [0.229, 0.224, 0.225]
else:
    h, w = 112, 112
    mean = [0.43216, 0.394666, 0.37645]
    std = [0.22803, 0.22145, 0.216989]
```

4. Define the image transformations for training:

```
import torchvision.transforms as transforms

train_transformer = transforms.Compose([
    transforms.Resize((h,w)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomAffine(degrees=0,
translate=(0.1,0.1)),
    transforms.ToTensor(),
    transforms.Normalize(mean, std),
])
```

5. Instantiate an object of the dataset class:

```
train_ds = VideoDataset(ids= train_ids, labels= train_labels,
transform= train_transformer)
print(len(train_ds))
```

The preceding snippet will print the following output:

632

6. Get a sample item from `train_ds`:

```
imgs, label = train_ds[1]
if len(imgs)>0:
    print(imgs.shape, label, torch.min(imgs), torch.max(imgs))
```

The preceding snippet will print the following output:

(`torch.Size([16, 3, 112, 112])`, 2, `tensor(-1.8952)`, `tensor(2.2908)`)

7. Let's display a few sample frames:

```
import matplotlib.pyplot as plt
%matplotlib inline

plt.figure(figsize=(10,10))
for ii,img in enumerate(imgs[::4]):
    plt.subplot(2,2,ii+1)
    plt.imshow(myutils.denormalize(img, mean, std))
    plt.title(label)
```

The preceding snippet will display the following image:



8. Define the transformations for the test dataset:

```
test_transformer = transforms.Compose([
    transforms.Resize((h,w)),
    transforms.ToTensor(),
    transforms.Normalize(mean, std),
])
```

9. Instantiate an object of the `VideoDataset` class as `test_ds`:

```
test_ds = VideoDataset(ids= test_ids, labels= test_labels,
transform= test_transformer)
print(len(test_ds))
```

The preceding snippet will print the following output:

71

10. Get a sample item from `test_ds`:

```
imgs, label = test_ds[1]
imgs.shape, label, torch.min(imgs), torch.max(imgs)
```

The preceding snippet will print the following output:

```
(torch.Size([16, 3, 112, 112]), 2, tensor(-1.8952), tensor(2.6627))
```

11. Let's display the sample item from `test_ds`:

```
plt.figure(figsize=(10,10))
for ii,img in enumerate(imgs[:4]):
    plt.subplot(2,2,ii+1)
    plt.imshow(myutils.denormalize(img, mean, std))
    plt.title(label)
```

The preceding snippet will display the following image:



In the next section, we will define the data loaders.

Defining the data loaders

As you know, we need data loaders to extract mini-batches for model training. Depending on the model type, we will define two data loaders:

1. Define the data loaders:

```
batch_size = 16
if model_type == "rnn":
    train_dl = DataLoader(train_ds, batch_size=batch_size,
                          shuffle=True, collate_fn=collate_fn_rnn)
```

```
    test_dl = DataLoader(test_ds, batch_size= 2*batch_size,
                         shuffle=False, collate_fn= collate_fn_rnn)
else:
    train_dl = DataLoader(train_ds, batch_size= batch_size,
                          shuffle=True, collate_fn=
    collate_fn_r3d_18)
    test_dl = DataLoader(test_ds, batch_size= 2*batch_size,
                         shuffle=False, collate_fn=
    collate_fn_r3d_18)
```

Here, `collate_fn_3dcnn` is defined as follows:

```
def collate_fn_3dcnn(batch):
    imgs_batch, label_batch = list(zip(*batch))
    imgs_batch = [imgs for imgs in imgs_batch if len(imgs)>0]
    label_batch = [torch.tensor(l) for l, imgs in zip(label_batch,
    imgs_batch) if len(imgs)>0]
    imgs_tensor = torch.stack(imgs_batch)
    imgs_tensor = torch.transpose(imgs_tensor, 2, 1)
    labels_tensor = torch.stack(label_batch)
    return imgs_tensor, labels_tensor
```

`collate_fn_rnn` is defined as follows:

```
def collate_fn_rnn(batch):
    imgs_batch, label_batch = list(zip(*batch))
    imgs_batch = [imgs for imgs in imgs_batch if len(imgs)>0]
    label_batch = [torch.tensor(l) for l, imgs in zip(label_batch,
    imgs_batch) if len(imgs)>0]
    imgs_tensor = torch.stack(imgs_batch)
    labels_tensor = torch.stack(label_batch)
    return imgs_tensor, labels_tensor
```

2. Now, set the model type to "3dcnn" and get a mini-batch from `train_dl`:

```
for xb,yb in train_dl:
    print(xb.shape, yb.shape)
    break
```

The preceding snippet will print the following output:

```
torch.Size([16, 3, 16, 112, 112]) torch.Size([16])
```

Repeat the preceding step, but this time set `model_type` to "rnn". You will see the following output:

```
torch.Size([16, 3, 16, 112, 112]) torch.Size([16])
```

In the next section, we will explain each step in detail.

How it works...

In *Preparing the data* subsection, we converted the videos into images. Since loading a video is a time-consuming process, we did this step in advance. Loading images is a lot faster than loading videos. In *step 1*, we got the list of action categories. As expected, there were 51 action categories. In *step 2*, we got the number of videos in each action category. As expected, there were more than 100 videos per class.

In *step 3*, we imported `myutils`. This utility file contains some of the helper functions that are required for this book. To save space, we put some of the less important helper functions in the `myutils.py` file. Once imported, we used the helper function defined in the file. The `get_frames` helper function loads a video from its filename and returns the specified number of frames. The `store_frames` helper function gets the frames and stores them in the given path. There is a technical tip to pay attention to here. The well-known OpenCV package loads images in BGR format, so we used `cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)` to convert images into RGB in the `get_frames` helper function. Conversely, OpenCV assumes BGR format when saving an image, so we used `cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)` in the `store_frames` helper function.



OpenCV loads and stores images in BGR format. You can use `cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)` to convert images into RGB.

In *step 4*, we looped over the videos, got 16 frames per videos, and stored them as `jpg` images.

In *Splitting the data* subsection, we got the list of video filenames and split them into two groups for training and test. In *step 1*, we imported `myutils`. Essential helper functions are in this utility file. We assume that you completed the previous steps and that the necessary images are now stored in the `hmdb51_jpg` folder. In *step 2*, we called the `get_vids` helper function.

Using this helper function, we got the list of video filenames. As you may recall, we extracted 16 frames per video. For each video, the 16 frames were stored in a folder that had the same name as its video filename. As we saw, there are 6,766 videos and labels. Each video has a text label corresponding to the activity of the video. Also, an excerpt from the `all_vids` and `all_labels` lists were printed.

In *step 3*, we defined a Python dictionary to hold the labels. Since the labels were in text format and we need a numerical equivalent for the next steps, we arbitrarily assigned a number to each action class. In *step 4*, we picked five action classes out of a total of 51 classes. This was done to simplify our problem. You can increase or decrease the number of classes. As we saw, when we filtered the videos to five classes, there were 703 videos left. This reduces the data size and the complexity of the problem. In *step 5*, we used the `StratifiedShuffleSplit` function from `sklearn` to split the data into training and test. We separated 10 percent for the test. As we saw, after splitting, 632 videos were in the training dataset and 71 videos were in the test dataset.

In *Defining the datasets* subsection, we created the dataset class. In *step 1*, we imported all the essential packages and set the random seed points for reproducibility. In *step 2*, we defined the dataset class, that is, `VideoDataset`. This class has three methods or functions.

The inputs to the `__init__` function are as follows:

- `ids`: List of video filenames
- `labels`: List of categorical labels corresponding to `ids`
- `transform`: Image transformations function

The input to the `__getitem__` function is as follows:

- `idx`: Path to the folder containing the 16 .jpg images of a video

In the function, we got the list of .jpg images and then loaded them as PIL images. Then, we performed image transformations on each image. Notice that we wanted to perform the same type of transformation on all 16 frames of a video. Thus, we set the random seed point every time we called the transformation. This will ensure that all 16 images will go through the same transformations.



To force the same image transformation on sequential images, set the random seed point to a fix number before calling the transformation.

In *step 3*, we defined a few parameters that were needed for image transformations. These include `h`, `w` to resize images and `mean`, `std` to normalize images. Notice that, depending on the model, the parameters were set differently. You can choose either "`3dcnn`" or "`rnn`" as the model type. Later, we will explain each model in detail in the *Defining the Model* recipe. For the "`rnn`" model, we resized the images to 224 by 224 while for the "`3dcnn`" model, we resized the images to 112 by 112. The reason we did this is that the model has different pre-training configurations.

In *step 4*, we defined the image transformations. Notice that in addition to resizing and normalizing images, we used two data augmentation transformations:

`RandomHorizontalFlip` and `RandomAffine`. In *step 5*, we instantiated an object of the `VideoDataset` class, that is, `train_ds`. As expected, there are 632 videos in the training dataset. In *step 6*, we got a sample item from `train_ds`. This is an exploratory step to make sure that the returned tensor is in the correct format. The returned tensor shape is in the shape of `[timesteps, 3, h, w]`, where `timesteps=16`, `h`, and `w` depend on `model_type`.

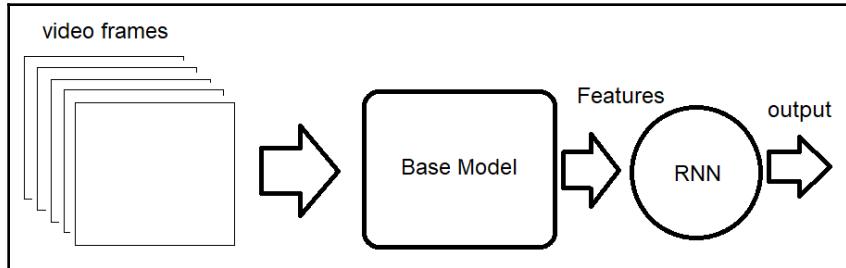
In *step 7*, we displayed a few sample frames from the returned tensor. In *step 8*, we defined transformations for the test dataset. Here, we don't need to perform data augmentation. In *step 9*, we instantiate an object of the `VideoDataset` class as `test_ds`. As expected, the test dataset contained 71 videos. In *step 10*, we get a sample item from `test_ds`. In *step 11*, we displayed a few frames of the sample tensor.

In the next section, we will define the model.

Defining the model

Compared to image classification, video classification is more complicated since we have to process several images at a time. As you may recall from [Chapter 2, Binary Image Classification](#), and [Chapter 3, Multi-Class Image Classification](#), we used a model based on a **two-dimensional convolutional neural network (2D-CNN)**. One simple approach would be to process images of a video one at a time using a 2D-CNN model and then average the outputs. However, this approach does not consider the temporal correlation between frames. Instead, we prefer to use a model that processes multiple images of a video in order to extract temporal correlation. To this end, we will use two different models for our video classification task.

The first model is based on RNN architecture. The goal of RNN models is to extract the temporal correlation between the images by keeping a memory of past images. The block diagram of the model is as follows:



As we can see, the images of a video are fed to a base model to extract high-level features. The features are then fed to an RNN layer and the output of the RNN layer is connected to a fully connected layer to get the classification output. The input to this model should be in the shape of [batch_size, timesteps, 3, height, width], where timesteps=16 is the number of frames per video. We will use one of the most popular models that has been pre-trained on the ImageNet dataset, called ResNet18, as the base model.

The second model is an 18-layer Resnet3D model, which was introduced in the following article: A Closer Look at Spatio-temporal Convolutions for Action Recognition (<https://arxiv.org/pdf/1711.11248.pdf>).

Let's call this model 3dcnn. The input to this model should be in the shape of [batch_size, 3, timesteps, height, width]. This model is available as a built-in model in the `torchvision.models.video` package. In this recipe, you will learn how to define two models for video classification.

How to do it...

Let's define two models for video classification:

1. Define Resnet18Rnn:

```
from torch import nn
class Resnt18Rnn(nn.Module):
    def __init__(self, params_model):
        super(Resnt18Rnn, self).__init__()
        num_classes = params_model["num_classes"]
        dr_rate= params_model["dr_rate"]
        pretrained = params_model["pretrained"]
```

```
rnn_hidden_size = params_model["rnn_hidden_size"]
rnn_num_layers = params_model["rnn_num_layers"]
```

The Resnet18Rnn class continues like so:

```
baseModel = models.resnet18(pretrained=pretrained)
num_features = baseModel.fc.in_features
baseModel.fc = Identity()
self.baseModel = baseModel
self.dropout= nn.Dropout(dr_rate)
self.rnn = nn.LSTM(num_features, rnn_hidden_size,
rnn_num_layers)
self.fc1 = nn.Linear(rnn_hidden_size, num_classes)

def forward(self, x):
    b_z, ts, c, h, w = x.shape
    ii = 0
    y = self.baseModel((x[:,ii]))
    output, (hn, cn) = self.rnn(y.unsqueeze(1))
    for ii in range(1, ts):
        y = self.baseModel((x[:,ii]))
        out, (hn, cn) = self.rnn(y.unsqueeze(1), (hn, cn))
        out = self.dropout(out[:, -1])
        out = self.fc1(out)
    return out
```

Also, define the Identity class as follows:

```
class Identity(nn.Module):
    def __init__(self):
        super(Identity, self).__init__()
    def forward(self, x):
        return x
```

2. Use a conditional statement to define either of the models:

```
from torchvision import models
from torch import nn

if model_type == "rnn":
    params_model={
        "num_classes": num_classes,
        "dr_rate": 0.1,
        "pretrained" : True,
        "rnn_num_layers": 1,
        "rnn_hidden_size": 100,}
    model = Resnt18Rnn(params_model)
else:
```

```
model = models.video.r3d_18(pretrained=True, progress=False)
num_features = model.fc.in_features
model.fc = nn.Linear(num_features, num_classes)
```

3. Let's test the model using some dummy input:

```
with torch.no_grad():
    if model_type=="rnn":
        x = torch.zeros(1, 16, 3, h, w)
    else:
        x = torch.zeros(1, 3, 16, h, w)
    y= model(x)
    print(y.shape)
```

The preceding snippet will print the following output:

```
torch.Size([1, 5])
```

4. Move the model to a CUDA device:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")
model = model.to(device)
```

5. Now, let's print the model:

```
print(model)
```

Depending on `model_type`, the corresponding model will be printed. An excerpt of the 3dcnn model is shown in the following output:

```
VideoResNet(
  (stem): BasicStem(
    (0): Conv3d(3, 64, kernel_size=(3, 7, 7), stride=(1, 2, 2),
padding=(1, 3, 3), bias=False)
    (1): BatchNorm3d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  ...
)
```

An excerpt of the `rnn` model is shown in the following output:

```
Resnt18Rnn(
  (baseModel): ResNet(
    (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
(relu) : ReLU(inplace=True)
(maxpool) : MaxPool2d(kernel_size=3, stride=2, padding=1,
dilation=1, ceil_mode=False)
...
```

In the next section, we will explain each step in detail.

How it works...

In *step 1*, we defined the RNN model class, that is, `Resnet18Rnn`. We used a Resnet18 model that had been pre-trained on the ImageNet dataset as the feature extractor. The extracted features were then fed to an RNN layer to extract the temporal correlation. The output of the RNN layer was fed to a fully connected layer to get the classification output. In *step 2*, we used an if condition to choose between one of the models. If `model_type` was set to "rnn", the RNN model was instantiated using the `Resnet18RNN` class; otherwise, the "3dcnn" model was defined using PyTorch's built-in models.

In *step 3*, we tested the defined model to make sure everything was correct. We passed some dummy input to the model and got the expected output. Use this step to debug your models before moving on to the next steps. In *step 4*, we defined a CUDA device and moved the model to the CUDA device. In *step 5*, we printed the model. Depending on `model_type`, the corresponding model was printed.

In the next section, we will train the model.

Training the model

So far, we've defined the dataset, data loaders, and the model. You may notice that this process was similar to that of image classification, but with some changes in the data format and model. It's not surprising that we can also use the same loss function and optimizer as defined in [Chapter 3, Multi-Class Image Classification](#). Also, for the training process, we will use the same stochastic gradient descent algorithm. To avoid repetition, we've put most of the training scripts in the `myutils.py` file that's provided as part of this book. Please see Chapter 3 for more details.

In the recipe, you will learn how to train a video classification model.

How to do it...

Let's define the loss function and optimizer and train the model:

1. Define the loss function, optimizer, and learning rate schedule:

```
from torch import optim
from torch.optim.lr_scheduler import CosineAnnealingLR,
ReduceLROnPlateau

loss_func = nn.CrossEntropyLoss(reduction="sum")
opt = optim.Adam(model.parameters(), lr=3e-5)
lr_scheduler = ReduceLROnPlateau(opt, mode='min', factor=0.5,
patience=5, verbose=1)
os.makedirs("./models", exist_ok=True)
```

2. Call the `train_val` helper function from `myutils` to train the model:

```
params_train={
    "num_epochs": 20,
    "optimizer": opt,
    "loss_func": loss_func,
    "train_dl": train_dl,
    "val_dl": test_dl,
    "sanity_check": True,
    "lr_scheduler": lr_scheduler,
    "path2weights": "./models/weights_"+model_type+".pt",
}
model, loss_hist, metric_hist = myutils.train_val(model, params_train)
```

After running the preceding snippet, training will start and you should see its progress printed on the screen.

3. After training is complete, plot the training progress:

```
myutils.plot_loss(loss_hist, metric_hist)
```

The preceding snippet will display a plot of loss and accuracy.

Once you've finished training the model, you can redo these steps to train the other model. You can change `model_type` to either "`rnn`" or "`3dcnn`" in the *Creating the dataset* recipe, in the *Defining the data loaders* subsection, and execute all the steps afterward. A better approach would be to restart your Jupyter Notebook after changing `model_type`.

How it works...

In *step 1*, we defined the loss function, optimizer, and the learning rate schedule. We used the same definition as Chapter 3, *Multi-Class Image Classification*. Please refer to that chapter for more details.

In *step 2*, we called the `train_val` helper function from `myutils`. This function was explained in detail in Chapter 3, *Multi-Class Image Classification*.

In *step 3*, we called the helper function defined in `myutils` to plot the training progress.

Deploying the video classification model

We've finished training two different models. Now, it's time to deploy the model on a video. We will follow the same principle of model deployment that we detailed in Chapter 3. To avoid repetition, we've put the required utility functions in the `myutils.py` file. To deploy the model in a separate script to the training script, we need to instantiate an object of the model class. You can do this by calling the `get_model` utility function defined in the `myutils.py` file. Then, we will load the trained weights into the model.

In this recipe, you will learn how to deploy our video classification model.

How to do it...

Let's instantiate an object of the model, load the pre-trained weights into the model, and deploy the model on a video:

1. Load one of the models:

```
import myutils

model_type = "rnn"
model = myutils.get_model(model_type = model_type, num_classes = 5)
model.eval();
```

In the preceding snippet, you can set `model_type` to `"3dcnn"` to load the second model.

2. Load the weights into the model:

```
import torch
device = torch.device("cuda:0" if torch.cuda.is_available() else
```

```
"cpu")  
  
path2weights = "./models/weights_"+model_type+".pt"  
model.load_state_dict(torch.load(path2weights))  
model.to(device);
```

3. Get the frames from a sample video:

```
path2video =  
"./data/hmdb51_org/brush_hair/April_09_brush_hair_u_nm_np1_ba_goo_0  
.avi"  
frames, v_len = myutils.get_frames(path2video, n_frames=16)  
len(frames), v_len
```

The preceding snippet will print the following output:

```
(16, 409)
```

4. Let's visualize a few frames:



5. Convert the frames into a tensor using the helper function defined in `myutils`:

```
imgs_tensor = myutils.transform_frames(frames, model_type)
print(imgs_tensor.shape, torch.min(imgs_tensor),
      torch.max(imgs_tensor))
```

The preceding snippet will print the following output:

```
torch.Size([1, 16, 3, 224]) tensor(-2.1179) tensor(2.6400)
```

6. Get the model's prediction:

```
with torch.no_grad():
    out = model(imgs_tensor.to(device)).cpu()
    print(out.shape)
    pred = torch.argmax(out).item()
    print(pred)
```

The preceding snippet will print the following output:

```
torch.Size([1, 5])
3
```

In the next section, we will explain each step in detail.

How it works...

In *step 1*, we called the `get_model` helper function defined in the `myutils.py` file to instantiate a model object. The inputs to the model are as follows:

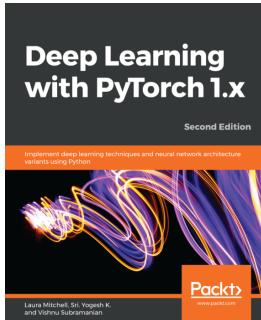
- `model_type`: Can be either "`rnn`" or "`3dcnn`".
- `num_classes`: Number of classification categories. The default is set to five.

In *step 2*, we loaded the pre-trained weights into the model. In *step 3*, we called the `get_frames` helper function defined in the `myutils.py` file to get the 16 frames of a video. In *step 4*, we displayed a few sample images.

In *step 5*, we called the `transform_frames` helper function defined in the `myutils.py` file to transform the frames into a PyTorch tensor. These transformations are the same ones that we defined in the *Creating the dataset* recipe, in the *Defining the PyTorch datasets* subsection. In *step 6*, we passed the tensor to the model and got its output.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

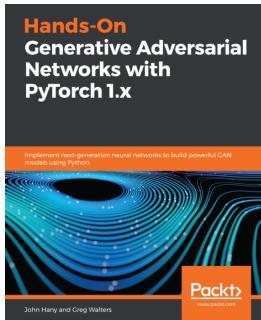


Deep Learning with PyTorch 1.x

Laura Mitchell, Sri. Yogesh K., Et al

ISBN: 978-1-83855-300-5

- Build text classification and language modeling systems using neural networks
- Implement transfer learning using advanced CNN architectures
- Use deep reinforcement learning techniques to solve optimization problems in PyTorch
- Mix multiple models for a powerful ensemble model
- Build image classifiers by implementing CNN architectures using PyTorch
- Get up to speed with reinforcement learning, GANs, LSTMs, and RNNs with real-world examples



Hands-On Generative Adversarial Networks with PyTorch 1.x

John Hany, Greg Walters

ISBN: 978-1-78953-051-3

- Implement PyTorch's latest features to ensure efficient model designing
- Get to grips with the working mechanisms of GAN models
- Perform style transfer between unpaired image collections with CycleGAN
- Build and train 3D-GANs to generate a point cloud of 3D objects
- Create a range of GAN models to perform various image synthesis operations
- Use SEGAN to suppress noise and improve the quality of speech audio

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

Albumentations

URL 140

Anaconda

installation link 10

installing 10, 11

Augmenter

URL 140

B

binary classification model

data, transforming 52, 53

binary cross-entropy (BCE) 296

built-in datasets

reference link 93

C

classification model

building 55, 56, 57, 58, 59, 60, 61, 62

COCO dataset

creating 171, 172, 173

COCO train2017 dataset 258

competition dataset

download link 42

Compute Unified Device Architecture (CUDA) 151

content and style images

loading 271, 272, 273, 274, 275, 276

convolutional neural networks (CNNs) 55, 152,

232

cross-entropy (CE) 262

CUDA device

models, moving to 29

custom COCO dataset

creating 173, 174, 175, 176, 177, 178, 179

custom dataset

creating 46, 47, 48, 49

custom datasets, for single-object segmentation

creating 222, 223, 226, 227, 228, 229, 230, 231, 232

data, augmenting 225

data, exploring 223, 224, 225

D

Darknet model 192, 193

data transformation

for single-object detection 130, 131, 132, 133, 134, 135, 136, 137

using 22, 23

data

transforming 180, 181, 182, 183

dataloaders

creating 24

defining, for training and validation of datasets 184, 185, 186, 187, 188, 189

dataset

exploring 42, 43, 44, 45, 46

loading 20, 21, 22

PyTorch tensors, wrapping into 24

splitting 49, 50, 51, 52

Deep Convolutional GAN (DCGAN)

about 287

URL 287

DeepLabV3Net101 258

E

encoder-decoder model

defining 232, 233, 234, 235, 236, 237

training 240, 241, 243, 244, 245

extracted features 152

F

Fast Gradient Sign (FGS) 304

FGS attack, implementation
dataset, loading 305
pre-trained model, loading 305, 306
FGS attack, on models
with adversarial examples 303, 304, 309, 310
FGS attack
implementing 307, 308, 309
using 311

G

GAN framework
training 297, 298, 299, 300, 301
GAN network
loss function, defining 296
optimizer, defining 296
Generative Adversarial Networks (GANs)
about 286
dataset, creating to train 287, 288, 289, 290
URL 286
generator and discriminator models
implementing 291, 292, 294, 295
generator model
deploying 301, 302

H

HMDB dataset
about 313
creating 313, 314, 315, 326, 327, 328
data loaders, defining 324, 325
data, preparing 315, 316
data, splitting 317, 318, 319
download link 314
PyTorch datasets, defining 319, 320, 321, 323, 324
reference link 313

I

iChallenge-AMD dataset
download link 122
exploring 122, 123, 124, 125, 126, 127, 128, 129
image recognition 41
ImageNet dataset
reference link 84
images

creating, by adjusting brightness 137, 138, 139, 140
imgaug
URL 140
installation
verifying 12, 13
Intersection over Union (IOU) 153, 207
IOU metric 153, 154, 155, 156, 157
defining 158

J

Jaccard index 153

L

linear layer 26
loss function
defining 32, 33, 34, 62, 63, 64
reference link 34, 64

M

mean square error (MSE) 153, 283
model summary
printing 30
reference link 30
model, for multi-class classification task
building 94, 95, 96, 97, 98, 99, 100
deploying 114, 115, 116, 117, 118, 119
loss function, defining 100, 102
optimizer, defining 103, 104, 105
training 105, 106, 107, 109, 110, 111, 112, 113, 114
model, for multi-object detection
training 208, 210, 211, 212
model, for single-object detection
creating 149, 150, 151, 152, 153
deploying 164, 165, 167, 169
training and evaluation 158, 159, 160, 161, 163, 164

model, on sample image
deploying 213, 214, 215, 216, 217, 218, 219, 220
model, on test_set images
deploying 245, 247, 248
models
building 26, 31, 32

defining, nn.Module used 28, 29
defining, nn.Sequential used 27
deploying 37, 38, 76, 77, 78, 79
evaluating 34, 35, 36, 39
inference, on test dataset 79, 80, 81, 82, 83
loading 36, 37
moving, to CUDA device 29
storing 36, 37
training 34, 35, 36, 39
training and evaluation 67, 68, 69, 70, 71, 72, 73, 74, 75
multi-class image classification
 about 84
 data, loading 85, 87, 88, 89, 91, 92, 93
 data, processing 85, 87, 88, 89, 91, 92, 93
multi-object segmentation
 loss function 262, 263, 264
 model, training on VOC dataset for 265, 266, 267, 268, 269
 optimizer 262, 263, 264

N

neural style transfer
 algorithm, running 280, 281
 implementing 276, 277, 282, 283, 284, 285
 loss function, defining 278, 279, 280
 optimizer, defining 280
 pretrained model, loading 277
nn.Module
 used, for defining models 28, 29
nn.Sequential
 used, for defining models 27
NumPy arrays
 converting, into PyTorch tensors 16, 17

O

object segmentation 221
optimizer 32, 33, 34, 65, 66, 153, 154, 155, 156, 157, 158
optimizer, for single-object segmentation
 defining 237, 238, 239, 240

P

pre-trained Model
 loading 306

PyTorch 1.0
 loss functions, reference link 102
PyTorch data tools
 used, for loading data 19, 25
 used, for processing data 19, 25
PyTorch Dataloader
 creating 53, 54, 55
PyTorch modules
 creating, for YOLO-v3 model 191
PyTorch tensors
 converting, into NumPy arrays 16
 moving, between devices 17, 18
 types and operations, reference link 18
 working with 14, 15, 18
 wrapping, into dataset 24
PyTorch
 installing 11, 12
 supported method, reference link 105
 URL 11

R

recurrent neural networks (RNNs) 314
ResNet paper 149

S

semantic segmentation model
 creating 258, 259, 260, 261, 262
 deploying 258, 259, 260, 261, 262
single-object detection
 custom datasets, creating 141, 142, 143, 144, 145, 146, 147, 148
 data transformation for 130, 131, 132, 133, 134, 135, 136, 137
 loss function, defining 153, 154, 155, 156, 157, 158
single-object segmentation
 loss function, defining 237, 238, 239, 240
smoothed-L1 loss 153
 reference link 153
 using 153
software tools and packages
 installing 10, 13, 14
Stochastic Gradient Descent (SGD) 32, 103, 208
style image
 reference link 271

style transfer services
reference link 285

T

tensor data type
about 15
modifying 15, 16
three-dimensional convolutional neural network (3D-CNN) 314
transfer learning 105, 106, 107, 109, 110, 111, 112, 113, 114
transformation functions
developing 180, 181, 182, 183
two-dimensional convolutional neural network (2D-CNN) model
about 328
defining 328, 329, 330, 331
working 332

V

video classification model
deploying 334, 335, 336

training 332, 333, 334
Visual Object Classes (VOC) 249
VOC dataset
reference link 251
VOC segmentation data
custom datasets, creating 250, 251, 252, 253, 254, 255, 256, 257, 258

W

WSL
reference link 171

Y

YOLO-v3 architecture
loss function, defining 199, 201, 202, 203, 204, 205, 206, 207, 208
YOLO-v3 model
configuration file, parsing 190
creating 189, 190, 194, 195, 197, 198, 199
Darknet model, defining 192, 193
PyTorch modules, creating 191
You Only Look Once (YOLO) 170