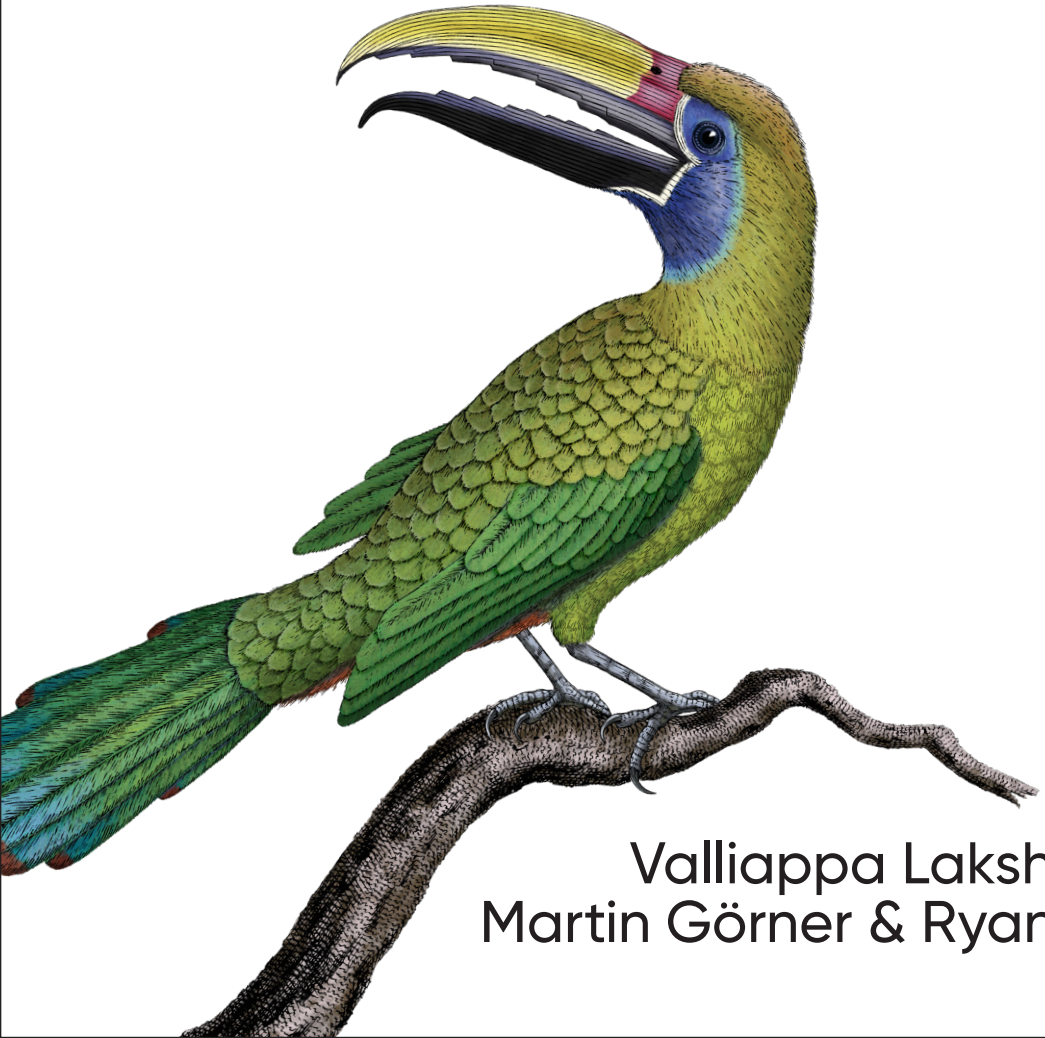
O'REILLY®

Practical Machine Learning for Computer Vision

End-to-End Machine Learning for Images



Valliappa Lakshmanan,
Martin Görner & Ryan Gillard

O'REILLY®

Practical Machine Learning for Computer Vision

This practical book shows you how to employ machine learning models to extract information from images. ML engineers and data scientists will learn proven ML techniques to solve a variety of image problems, including classification, object detection, autoencoders, image generation, counting, and captioning. This book provides a great introduction to end-to-end deep learning: dataset creation, data preprocessing, model design, model training, evaluation, deployment, and interpretability.

Google engineers Valliappa Lakshmanan, Martin Görner, and Ryan Gillard show you how to develop accurate and explainable computer vision ML models and put them into large-scale production using robust ML architecture in a flexible and maintainable way. You'll learn how to design, train, evaluate, and predict with models written in TensorFlow and Keras.

You'll learn how to:

- Design ML architecture for computer vision tasks
- Select a model (such as ResNet, SqueezeNet, or EfficientNet) appropriate to your task
- Create an end-to-end ML pipeline to train, evaluate, deploy, and explain your model
- Preprocess images for data augmentation and to support learnability
- Incorporate explainability and responsible AI best practices
- Deploy image models as web services or on-edge devices
- Monitor and manage ML models

"This book provides a thorough introduction to the current state of the art for deep computer vision, along with battle-tested best practices for building end-to-end production systems in Keras to solve real-world problems."

—**François Chollet**
Deep learning researcher and creator of Keras

Valliappa (Lak) Lakshmanan is the director of analytics and AI solutions at Google Cloud, where he leads a team building cross-industry solutions to business problems.

Martin Görner is a product manager for Keras/TensorFlow focused on improving the developer experience when using state-of-the-art models.

Ryan Gillard is an AI engineer in Google Cloud's Professional Services organization, where he builds ML models for a wide variety of industries. He started his career as a research scientist in the hospital and healthcare industry.

DATA | DATA SCIENCE | DATA ANALYTICS | MACHINE LEARNING
DEEP LEARNING | PYTHON MACHINE LEARNING

US $79.99 CAN $105.99

ISBN: 978-1-098-10236-4

Twitter: @oreillymedia
facebook.com/oreilly

Practical Machine Learning for Computer Vision

End-to-End Machine Learning for Images

Valliappa Lakshmanan, Martin Görner, and Ryan Gillard

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Practical Machine Learning for Computer Vision
by Valliappa Lakshmanan, Martin Görner, and Ryan Gillard

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisition Editor: Rebecca Novack
Development Editor: Amelia Blevins and Shira Evans
Production Editor: Katherine Tozer
Copyeditor: Rachel Head
Proofreader: Piper Editorial Consulting, LLC
Indexer: Ellen Troutman-Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Robert Romano

July 2021: First Edition

Revision History for the First Edition
2021-07-21: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098102364* for release details.

978-1-098-10236-4

[LSI]

Table of Contents

Preface

Machine learning on images is revolutionizing healthcare, manufacturing, retail, and many other sectors. Many previously difficult problems can now be solved by training machine learning (ML) models to identify objects in images. Our aim in this book is to provide intuitive explanations of the ML architectures that underpin this fast-advancing field, and to provide practical code to employ these ML models to solve problems involving classification, measurement, detection, segmentation, representation, generation, counting, and more.

Image classification is the "hello world" of deep learning. Therefore, this book also provides a practical end-to-end introduction to deep learning. It can serve as a stepping stone to other deep learning domains, such as natural language processing.

You will learn how to design ML architectures for computer vision tasks and carry out model training using popular, well-tested prebuilt models written in TensorFlow and Keras. You will also learn techniques to improve accuracy and explainability. Finally, this book will teach you how to design, implement, and tune end-to-end ML pipelines for image understanding tasks.

Who Is This Book For?

The primary audience for this book is software developers who want to do machine learning on images. It is meant for developers who will use TensorFlow and Keras to solve common computer vision use cases.

The methods discussed in the book are accompanied by code samples available at *https://github.com/GoogleCloudPlatform/practical-ml-vision-book*. Most of this book involves open source TensorFlow and Keras and will work regardless of whether you run the code on premises, in Google Cloud, or in some other cloud.

Developers who wish to use PyTorch will find the textual explanations useful, but will probably have to look elsewhere for practical code snippets. We do welcome

contributions of PyTorch equivalents of our code samples; please make a pull request to our GitHub repository.

How to Use This Book

We recommend that you read this book in order. Make sure to read, understand, and run the accompanying notebooks in the book's GitHub repository—you can run them in either Google Colab or Google Cloud's Vertex Notebooks. We suggest that after reading each section of the text you try out the code to be sure you fully understand the concepts and techniques that are introduced. We strongly recommend completing the notebooks in each chapter before moving on to the next chapter.

Google Colab is free and will suffice to run most of the notebooks in this book; Vertex Notebooks is more powerful and so will help you run through the notebooks faster. The more complex models and larger datasets of Chapters 3, 4, 11, and 12 will benefit from the use of Google Cloud TPUs. Because all the code in this book is written using open source APIs, the code *should* also work in any other Jupyter environment where you have the latest version of TensorFlow installed, whether it's your laptop, or Amazon Web Services (AWS) Sagemaker, or Azure ML. However, we haven't tested it in those environments. If you find that you have to make any changes to get the code to work in some other environment, please do submit a pull request in order to help other readers.

The code in this book is made available to you under an Apache open source license. It is meant primarily as a teaching tool, but can serve as a starting point for your production models.

Organization of the Book

The remainder of this book is organized as follows:

- In Chapter 2, we introduce machine learning, how to read in images, and how to train, evaluate, and predict with ML models. The models we cover in Chapter 2 are generic and thus don't work particularly well on images, but the concepts introduced in this chapter are essential for the rest of the book.
- In Chapter 3, we introduce some machine learning models that do work well on images. We start with transfer learning and fine-tuning, and then introduce a variety of convolutional models that increase in sophistication as we get further and further into the chapter.
- In Chapter 4, we explore the use of computer vision to address object detection and image segmentation problems. Any of the backbone architectures introduced in Chapter 3 can be used in Chapter 4.

- In Chapters 5 through 9, we delve into the details of creating production computer vision machine learning models. We go though the standard ML pipeline stage by stage, looking at dataset creation in Chapter 5, preprocessing in Chapter 6, training in Chapter 7, monitoring and evaluation in Chapter 8, and deployment in Chapter 9. The methods discussed in these chapters are applicable to any of the model architectures and use cases discussed in Chapters 3 and 4.
- In Chapter 10, we address three up-and-coming trends. We connect all the steps covered in Chapters 5 through 9 into an end-to-end, containerized ML pipeline, then we try out a no-code image classification system that can serve for quick prototyping and as a benchmark for more custom models. Finally, we show how to build explainability into image model predictions.
- In Chapters 11 and 12, we demonstrate how the basic building blocks of computer vision are used to solve a variety of problems, including image generation, counting, pose detection, and more. Implementations are provided for these advanced use cases as well.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, data types, environment variables, statements, and keywords.

`Constant width bold`
: Used for emphasis in code snippets, and to show command or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element signifies a warning.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/GoogleCloudPlatform/practical-ml-vision-book*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Practical Machine Learning for Computer Vision*, by Valliappa Lakshmanan, Martin Görner, and Ryan Gillard. Copyright 2021 Valliappa Lakshmanan, Martin Görner, and Ryan Gillard, 978-1-098-10236-4."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning

paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/practical-ml-4-computer-vision*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

We are very thankful to Salem Haykal and Filipe Gracio, our superstar reviewers who reviewed every chapter in this book—their eye for detail can be felt throughout. Thanks also to the O'Reilly technical reviewers Vishwesh Ravi Shrimali and Sanyam Singhal for suggesting the reordering that improved the organization of the book. In addition, we would like to thank Rajesh Thallam, Mike Bernico, Elvin Zhu, Yuefeng Zhou, Sara Robinson, Jiri Simsa, Sandeep Gupta, and Michael Munn for reviewing chapters that aligned with their areas of expertise. Any remaining errors are ours, of course.

We would like to thank Google Cloud users, our teammates, and many of the cohorts of the Google Cloud Advanced Solutions Lab for pushing us to make our explanations crisper. Thanks also to the TensorFlow, Keras, and Google Cloud AI engineering teams for being thoughtful partners.

Our O'Reilly team provided critical feedback and suggestions. Rebecca Novack suggested updating an earlier O'Reilly book on this topic, and was open to our recommendation that a practical computer vision book would now involve machine learning and so the book would require a complete rewrite. Amelia Blevins, our editor at O'Reilly, kept us chugging along. Rachel Head, our copyeditor, and Katherine Tozer, our production editor, greatly improved the clarity of our writing.

Finally, and most importantly, thanks also to our respective families for their support.

— Valliappa Lakshmanan, Bellevue, WA
Martin Görner, Bellevue, WA
Ryan Gillard, Pleasanton, CA

CHAPTER 1

Machine Learning for Computer Vision

Imagine that you are sitting in a garden, observing what's going on around you. There are two systems in your body that are at work: your eyes are acting as sensors and creating representations of the scene, while your cognitive system is making sense of what your eyes are seeing. Thus, you might see a bird, a worm, and some movement and realize that the bird has walked down the path and is eating a worm (see Figure 1-1).

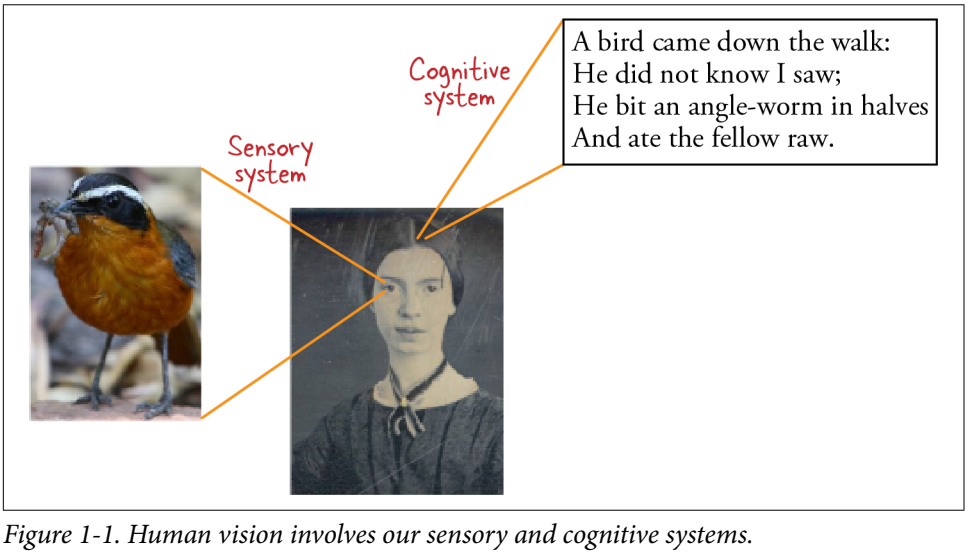


Figure 1-1. Human vision involves our sensory and cognitive systems.

Computer vision tries to imitate human vision capabilities by providing methods for image formation (mimicking the human *sensory* system) and machine perception (mimicking the human *cognitive* system). Imitation of the human sensory system is focused on hardware and on the design and placement of sensors such as cameras.

The modern approach to imitating the human cognitive system consists of machine learning (ML) methods that are used to extract information from images. It is these methods that we cover in this book.

When we see a photograph of a daisy, for example, our human cognitive system is able to recognize it as a daisy (see Figure 1-2). The machine learning models for image classification that we build in this book imitate this human capability by starting from photographs of daisies.

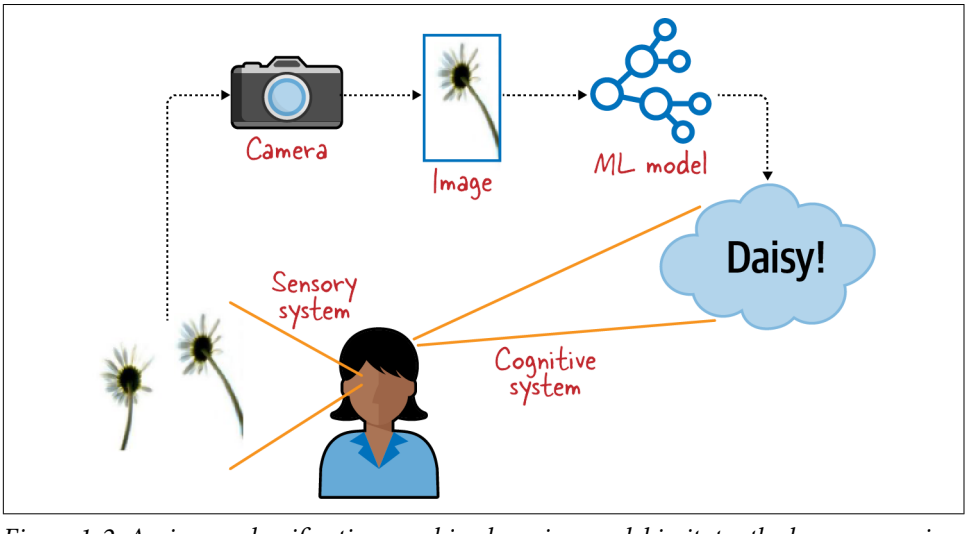


Figure 1-2. An image classification machine learning model imitates the human cognitive system.

Machine Learning

If you were reading a book on computer vision in the early 2010s, the methods used to extract information from photographs would not have involved machine learning. Instead, you would have been learning about denoising, edge finding, texture detection, and morphological (shape-based) operations. With advancements in artificial intelligence (more specifically, advances in machine learning), this has changed.

Artificial intelligence (AI) explores methods by which computers can mimic human capabilities. *Machine learning* is a subfield of AI that teaches computers to do this by showing them a large amount of data and instructing them to learn from it. *Expert systems* is another subfield of AI—expert systems teach computers to mimic human capabilities by programming the computers to follow human logic. Prior to the 2010s, computer vision tasks like image classification were commonly done by building bespoke image filters to implement the logic laid out by experts. Nowadays, image

classification is achieved through convolutional networks, a form of deep learning (see Figure 1-3).

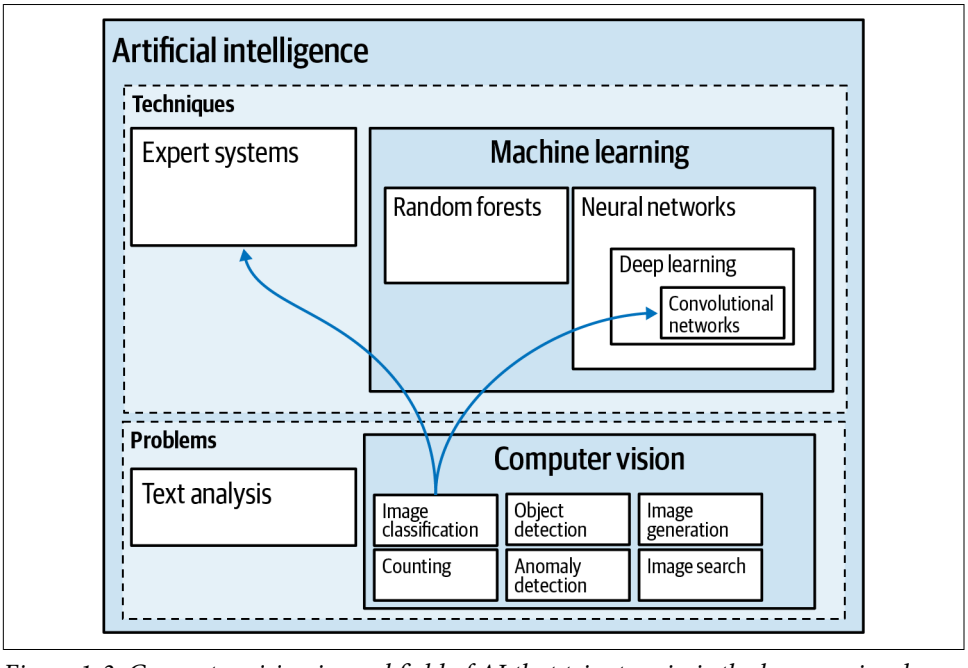


Figure 1-3. Computer vision is a subfield of AI that tries to mimic the human visual system; while it used to rely on an expert systems approach, today it's done with machine learning.

Take, for example, the image of the daisy in Figure 1-2. A machine learning approach teaches a computer to recognize the type of flower in an image by showing the computer lots of images along with their *labels* (or correct answers). So, we'd show the computer lots of images of daisies, lots of images of tulips, and so on. Based on such a *labeled training dataset,* the computer learns how to classify an image that it has not encountered before. How this happens is discussed in Chapters 2 and 3.

In an expert system approach, on the other hand, we would start by interviewing a human botanist on how they classify flowers. If the botanist explained that *bellis perennis* (the scientific name for a daisy) consists of white elongated petals around a yellow center and green, rounded leaves, we would attempt to devise image processing filters to match these criteria. For example, we'd look for the prevalence of white, yellow, and green in the image. Then we'd devise edge filters to identify the borders of the leaves and matched morphological filters to see if they match the expected rounded shape. We might smooth the image in HSV (hue, saturation, value) space to determine the color of the center of the flower as compared to the color of the petals. Based on these criteria, we might come up with a score for an image that rates the

likelihood that it is a daisy. Similarly, we'd design and apply different sets of rules for roses, tulips, sunflowers, and so on. To classify a new image, we'd pick the category whose score is highest for that image.

This description illustrates the considerable bespoke work that was needed to create image classification models. This is why image classification used to have limited applicability.

That all changed in 2012 with the publication of the AlexNet paper. The authors—Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton—were able to greatly outperform any existing image classification method by applying convolutional networks (covered in Chapter 3) to the benchmark dataset used in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). They achieved a top-5[1] error of 15.3%, while the error rate of the runner-up was over 26%. Typical improvements in competitions like this are on the order of 0.1%, so the improvement that AlexNet demonstrated was one hundred times what most people expected! This was an attention-grabbing performance.

Neural networks had been around since the 1970s, and convolutional neural networks (CNNs) themselves had been around for more than two decades by that point —Yann LeCun introduced the idea in 1989. So what was new about AlexNet? Four things:

Graphics processing units (GPUs)
: Convolutional neural networks are a great idea, but they are computationally very expensive. The authors of AlexNet implemented a convolutional network on top of the graphics rendering libraries provided by special-purpose chips called GPUs. GPUs were, at the time, being used primarily for high-end visualization and gaming. The paper grouped the convolutions to fit the model across two GPUs. GPUs made convolutional networks feasible to train (we'll talk about distributing model training across GPUs in Chapter 7).

Rectified linear unit (ReLU) activation
: AlexNet's creators used a non-saturating activation function called ReLU in their neural network. We'll talk more about neural networks and activation functions in Chapter 2; for now, it's sufficient to know that using a piecewise linear non-saturating activation function enabled their model to converge much faster.

1 *Top-5 accuracy* means that we consider the model to be correct if it returns the correct label for an image within its top five results.

Regularization
: The problem with ReLUs—and the reason they hadn't been used much until 2012—was that, because they didn't saturate, the neural network's weights became numerically unstable. The authors of AlexNet used a regularization technique to keep the weights from becoming too large. We'll discuss regularization in Chapter 2 too.

Depth
: With the ability to train faster, they were able to train a more complex model that had more neural network layers. We say a model with more layers is *deeper*; the importance of depth will be discussed in Chapter 3.

It is worth recognizing that it was the increased depth of the neural network (allowed by the combination of the first three ideas) that made AlexNet world-beating. That CNNs could be sped up using GPUs had been proven in 2006. The ReLU activation function itself wasn't new, and regularization was a well-known statistical technique. Ultimately, the model's exceptional performance was due to the authors' insight that they could combine all of these to train a deeper convolutional neural network than had been done before.

Depth is so important to the resurging interest in neural networks that the whole field has come to be referred to as *deep learning*.

Deep Learning Use Cases

Deep learning is a branch of machine learning that uses neural networks with many layers. Deep learning outperformed the previously existing methods for computer vision, and has now been successfully applied to many other forms of unstructured data: video, audio, natural language text, and so on.

Deep learning gives us the ability to extract information from images without having to create bespoke image processing filters or code up human logic. When doing image classification using deep learning, we need hundreds or thousands or even millions of images (the more, the better), for which we know the correct label (like "tulip" or "daisy"). These labeled images can be used to train an image classification deep learning model.

As long as you can formulate a task in terms of learning from data, it is possible to use computer vision machine learning methods to address the problem. For example, consider the problem of optical character recognition (OCR)—taking a scanned image and extracting the text from it. The earliest approaches to OCR involved teaching the computer to do pattern matching against what individual letters look like. This turns out to be a challenging approach, for various reasons. For example:

- There are many fonts, so a single letter can be written in many ways.
- Letters come in different sizes, so the pattern matching has to be scale-invariant.
- Bound books cannot be laid flat, so the scanned letters are distorted.
- It is not enough to recognize individual letters; we need to extract the entire text. The rules of what forms a word, a line, or a paragraph are complex (see Figure 1-4).

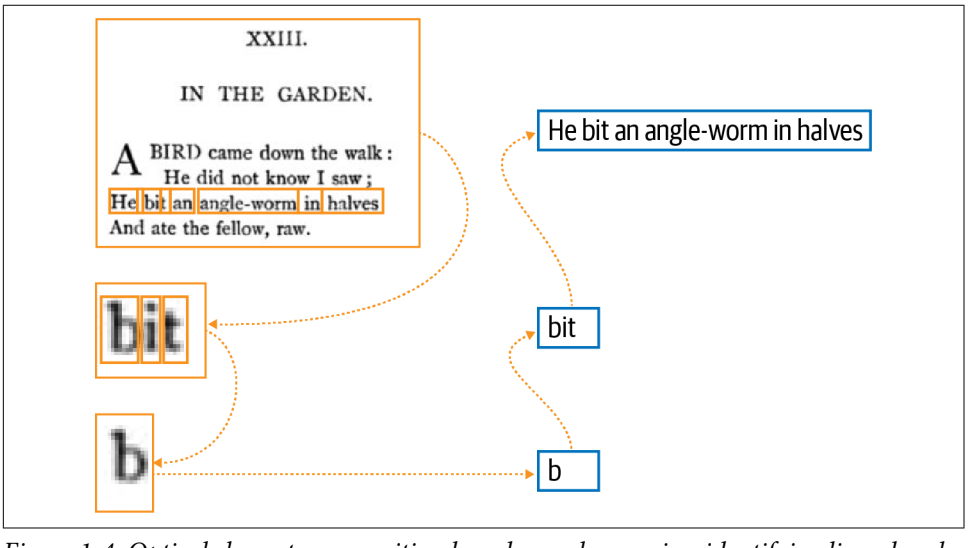


Figure 1-4. Optical character recognition based on rules requires identifying lines, breaking them into words, and then identifying the component letters of each word.

On the other hand, with the use of deep learning, OCR can be quite easily formulated as an image classification system. There are many books that have already been digitized, and it's possible to train the model by showing it a scanned image from a book and using the digitized text as a label.

Computer vision methods provide solutions for a variety of real-world problems. Besides OCR, computer vision methods have been successfully applied to medical diagnosis (using images such as X-rays and MRIs), automating retail operations (such as reading QR codes, recognizing empty shelves, checking the quality of vegetables, etc.), surveillance (monitoring crop yield from satellite images, monitoring wildlife cameras, intruder detection, etc.), fingerprint recognition, and automotive safety (following cars at a safe distance, identifying changes in speed limits from road signs, self-parking cars, self-driving cars, etc.).

Computer vision has found use in many industries. In government, it has been used for monitoring satellite images, in building smart cities, and in customs and security

inspections. In healthcare, it has been used to identify eye disease and to find early signs of cancer from mammograms. In agriculture, it has been used to spot malfunctioning irrigation pumps, assess crop yields, and identify leaf disease. In manufacturing, it finds a use on factory floors for quality control and visual inspection. In insurance, it has been used to automatically assess damage to vehicles after an accident.

Summary

Computer vision helps computers understand the content of digital images such as photographs. Starting with a seminal paper in 2012, deep learning approaches to computer vision have become wildly successful. Nowadays, we find successful uses of computer vision across a large number of industries.

We'll start our journey in Chapter 2 by creating our first machine learning models.

CHAPTER 2

ML Models for Vision

In this chapter, you will learn how to represent images and train basic machine learning models to classify images. You will discover that the performance of linear and fully connected neural networks is poor on images. However, along the way, you will learn how to use the Keras API to implement ML primitives and train ML models.

The code for this chapter is in the *02_ml_models* folder of the book's GitHub repository. We will provide file names for code samples and notebooks where applicable.

A Dataset for Machine Perception

For the purposes of this book, it will be helpful if we take a single practical problem and build a variety of machine learning models to solve it. Assume that we have collected and labeled a dataset of nearly four thousand photographs of flowers. There are five types of flowers in the *5-flowers* dataset (see Figure 2-1), and each image in the dataset has already been labeled with the type of flower it depicts.

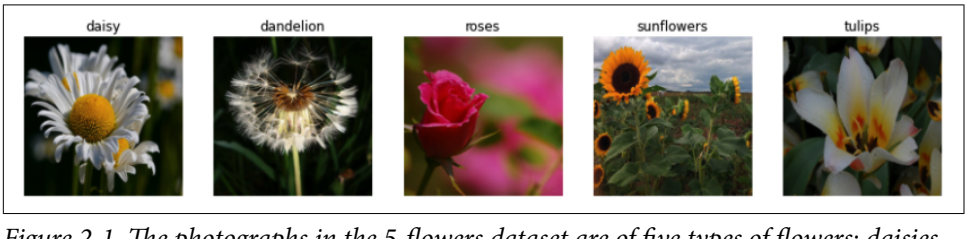


Figure 2-1. The photographs in the 5-flowers dataset are of five types of flowers: daisies, dandelions, roses, sunflowers, and tulips.

Suppose we want to create a computer program that will, when provided an image, tell us what type of flower is in the image. We are asking the machine learning model to learn to perceive what's in the image, so you might see this type of task called *machine perception*. Specifically, the type of perception is analogous to human sight, so the problem is termed *computer vision*, and in this case we will solve it through image classification.

5-Flowers Dataset

The 5-flowers dataset was created by Google and placed in the public domain with a Creative Commons license. It is published as a TensorFlow dataset and available in a public Google Cloud Storage bucket (`gs://cloud-ml-data/`) in the form of JPEG files. This makes the dataset both realistic (it consists of JPEG photographs of the sort collected by off-the-shelf cameras) and readily accessible. Therefore, we will use it as an ongoing example in this book.

An Example, but Not a Template

The 5-flowers dataset is a great dataset to learn with, but you should not use it as the template for how you create a training dataset. There are several factors that make the 5-flowers dataset subpar from the perspective of serving as a template:

Quantity
: To train ML models from scratch, you'll typically need to collect millions of images. There are alternative approaches that work with fewer images, but you should attempt to collect the largest dataset that is practical (and ethical!).

Data format
: Storing the images as individual JPEG files is very inefficient because most of your model training time will be spent waiting for data to be read. It is better to use TensorFlow Record format.

Content
: The dataset itself consists of *found* data—images that were not explicitly collected for the classification task. You should, if your problem domain allows, collect data more purposefully. More on this shortly.

Labeling
: The labeling of images is a topic in and of itself. This dataset was manually labeled. This can become impractical for larger datasets.

We will discuss these factors and provide best practices for how to design, collect, organize, store, and label data throughout the book.

In Figure 2-2, you can see several of the tulip photographs. Note that they range from close-up photographs to photographs of fields of tulips. All of these are photographs that a human would have no problem labeling as tulips, but it's a difficult problem for us to capture using simple rules—if we were to say a tulip is an elongated flower, for example, only the first and fourth images would qualify.

Figure 2-2. These five photographs of tulips vary widely in terms of zoom, color of the tulips, and what's in the frame.

Standardize Image Collection

We are choosing to address a hard problem where the flower images are all collected in real-world conditions. However, in practice, you can often make a machine perception problem easier by standardizing how the images are collected. For example, you could specify that your images have to be collected in controlled conditions, with flat lighting, and at a consistent zoom. This is common in the manufacturing industry—factory conditions can be precisely specified. It is also common to design the scanner in such a way that an object can be placed in only one orientation. As a machine learning practitioner, you should be on the lookout for ways to make machine perception problems easier. This is not cheating—it's the smart thing to do to set yourself up for success.

Keep in mind, however, that your training dataset has to reflect the conditions under which your model will be required to make predictions. If your model is trained only on photographs of flowers taken by professional photographers, it will probably do poorly on photographs taken by amateurs whose lighting, zoom, and framing choices are likely to be different.

Reading Image Data

To train image models, we need to read image data into our programs. There are four steps to reading an image in a standard format like JPEG or PNG and getting it ready to train machine learning models with it (the complete code is available in *02a_machine_perception.ipynb* in the GitHub repository for the book):

```
import tensorflow as tf
def read_and_decode(filename, reshape_dims):
    # 1. Read the file.
    img = tf.io.read_file(filename)
    # 2. Convert the compressed string to a 3D uint8 tensor.
    img = tf.image.decode_jpeg(img, channels=3)
    # 3. Convert 3D uint8 to floats in the [0,1] range.
    img = tf.image.convert_image_dtype(img, tf.float32)
    # 4. Resize the image to the desired size.
    return tf.image.resize(img, reshape_dims)
```

We first read the image data from persistent storage into memory as a sequence of bytes:

```
img = tf.io.read_file(filename)
```

The variable `img` here is a tensor (see "What's a Tensor?" on page 13) that contains an array of bytes. We parse these bytes to convert them into the pixel data—this is also called *decoding* the data because image formats like JPEG require you to decode the pixel values from lookup tables:

```
img = tf.image.decode_jpeg(img, channels=3)
```

Here, we specify that we want only the three color channels (red, green, and blue) from the JPEG image and not the opacity, which is the fourth channel. The channels you have available depend on the file itself. Grayscale images may have only one channel.

The pixels will consist of RGB values that are of type `uint8` and are in the range [0,255]. So, in the third step, we convert them to floats and scale the values to lie in the range [0,1]. This is because machine learning optimizers are tuned to work well with small numbers:

```
img = tf.image.convert_image_dtype(img, tf.float32)
```

Finally, we resize the image to the desired size. Machine learning models are built to work with inputs of known sizes. Since images in the real world are likely to come in arbitrary sizes, you might have to shrink, crop, or expand them to fit your desired size. For example, to resize the image to be 256 pixels wide and 128 pixels tall, we'd specify:

```
tf.image.resize(img,[256, 128])
```

In Chapter 6, we'll see that this method does not preserve the aspect ratio and we'll look at other options to resize images.

What's a Tensor?

In mathematics, a 1D array is called a vector, and a 2D array is called a matrix. A *tensor* is an array that could have any number of dimensions (the number of dimensions is called the *rank*). A matrix with 12 rows and 18 columns is said to have a *shape* of (12, 18) and a rank of 2. So, a tensor can have arbitrary shape.

The common array math library in Python is called `numpy`. You can use this library to create *n*-dimensional arrays, but the problem is that they are not hardware-accelerated. For example, this is a 1D array whose shape is (4):

```
x = np.array([2.0, 3.0, 1.0, 0.0])
```

whereas this is a 5D array of zeros (note that there are five numbers in the shape):

```
x5d = np.zeros(shape=(4, 3, 7, 8, 3))
```

To obtain hardware acceleration using TensorFlow, you can convert either `numpy` array into a tensor, which is how TensorFlow represents arrays, using:

```
tx = tf.convert_to_tensor(x, dtype=tf.float32)
```

And you can convert a tensor back into a `numpy` array using:

```
x = tx.numpy()
```

Mathematically, `numpy` arrays and TensorFlow tensors are the same thing. However, there is an important practical difference—all `numpy` arithmetic is done on the CPU, while the TensorFlow code runs on a GPU if one is available. Thus, doing:

```
x = x * 0.3
```

will typically be less efficient than:

```
tx = tx * 0.3
```

In general, the more you can do using TensorFlow operations, the more efficient your program will be. It is also more efficient if you *vectorize* your code (to process batches of images) so that you are carrying out a single in-place tensor operation instead of a bunch of tiny little scalar operations.

These steps are not set in stone. If your input data consists of remotely sensed images from a satellite that are provided in a band interleaved format or brain scan images provided in Digital Imaging and Communications in Medicine (DICOM) format, you obviously wouldn't decode those using `decode_jpeg()`. Similarly, you may not always resize the data. In some instances, you might choose to crop the data to the desired size or pad it with zeros. In other cases, you might resize, keeping the aspect ratio constant, and then pad the remaining pixels. These preprocessing operations are discussed in Chapter 6.

Visualizing Image Data

Always visualize a few of the images to ensure that you are reading the data correctly—a common mistake is to read the data in such a way that the images are rotated or mirrored. Visualizing the images is also useful to get a sense of how challenging a machine perception problem is.

We can use Matplotlib's `imshow()` function to visualize an image, but in order to do so we must first convert the image, which is a TensorFlow tensor, into a `numpy` array using the `numpy()` function.

```
def show_image(filename):
    img = read_and_decode(filename, [IMG_HEIGHT, IMG_WIDTH])
    plt.imshow(img.numpy());
```

Trying it out on one of our daisy images, we get what's shown in Figure 2-3.

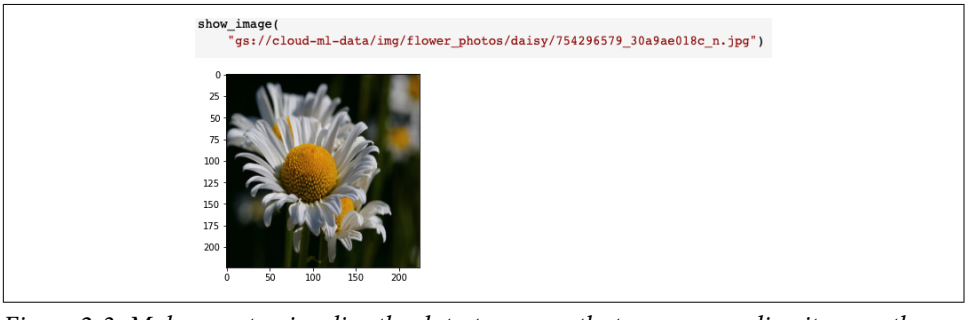


Figure 2-3. Make sure to visualize the data to ensure that you are reading it correctly.

Notice from Figure 2-3 that the filename contains the type of flower (daisy). This means we can use wildcard matching with TensorFlow's `glob()` function to get, say, all the tulip images:

```
tulips = tf.io.gfile.glob(
    "gs://cloud-ml-data/img/flower_photos/tulips/*.jpg")
```

The result of running this code and visualizing a panel of five tulip photographs was shown in Figure 2-2.

Reading the Dataset File

We now know how to read an image. In order to train a machine model, though, we need to read many images. We also have to obtain the labels for each of the images. We could obtain a list of all the images by carrying out a wildcard match using `glob()`:

```
tf.io.gfile.glob("gs://cloud-ml-data/img/flower_photos/*/*.jpg")
```

Then, knowing that the images in our dataset have a naming convention, we could take a filename and extract the label using string operations. For example, we can remove the prefix using:

```
basename = tf.strings.regex_replace(
    filename,
    "gs://cloud-ml-data/img/flower_photos/", "")
```

and get the category name using:

```
label = tf.strings.split(basename, '/')[0]
```

As usual, please refer to the GitHub repository for this book for the full code.

However, for reasons of generalization and reproducibility (explained further in Chapter 5), it is better to set aside in advance the images that we will retain for evaluation. That has already been done in the 5-flowers dataset, and the images to use for training and evaluation are listed in two files in the same Cloud Storage bucket as the images:

```
gs://cloud-ml-data/img/flower_photos/train_set.csv
gs://cloud-ml-data/img/flower_photos/eval_set.csv
```

These are comma-separated values (CSV) files where each line contains a filename followed by the label.

One way to read a CSV file is to read in text lines using `TextLineDataset`, passing in a function to handle each line as it is read through the `map()` function:

```
dataset = (tf.data.TextLineDataset(
    "gs://cloud-ml-data/img/flower_photos/train_set.csv").
    map(parse_csvline))
```

We are using the `tf.data` API, which makes it possible to handle large amounts of data (even if it doesn't all fit into memory) by reading only a handful of data elements at a time, and performing transformations as we are reading the data. It does this by using an abstraction called `tf.data.Dataset` to represent a sequence of elements. In our pipeline, each element is a training example that contains two tensors. The first tensor is the image and the second is the label. Many types of `Datasets` correspond to many different file formats. We're using `TextLineDataset`, which reads text files and assumes that each line is a different element.

`parse_csvline()` is a function that we supply in order to parse the line, extract the filename of the image, read the image, and return the image and its label:

```
def parse_csvline(csv_row):
    record_defaults = ["path", "flower"]
    filename, label = tf.io.decode_csv(csv_row, record_defaults)
    img = read_and_decode(filename, [IMG_HEIGHT, IMG_WIDTH])
    return img, label
```

The `record_defaults` that are passed into the `parse_csvline()` function specify what TensorFlow needs to replace in order to handle a line where one or more values are missing.

To verify that this code works, we can print out the average pixel value for each channel of the first three images in the training dataset:

```
for img, label in dataset.take(3):
    avg = tf.math.reduce_mean(img, axis=[0, 1])
    print(label, avg)
```

In this code snippet, the `take()` method truncates the dataset to three items. Notice that because `decode_csv()` returns a tuple `(img, label)`, that's what we obtain when we iterate through the dataset. Printing out the entire image is a terrible idea, so we are printing out the average pixel value in the image using `tf.reduce_mean()`.

The first line of the result is (with line breaks added for readability):

```
tf.Tensor(b'daisy', shape=(), dtype=string)
tf.Tensor([0.3588961  0.36257887 0.26933077],
          shape=(3,), dtype=float32)
```

Note that the label is a string tensor and the average is a 1D tensor of length 3. Why did we get a 1D tensor? That's because we passed in an `axis` parameter to `reduce_mean()`:

```
avg = tf.math.reduce_mean(img, axis=[0, 1])
```

Had we not supplied an axis, then TensorFlow would have computed the mean along all the dimensions and returned a scalar value. Recall that the shape of the image is `[IMG_HEIGHT, IMG_WIDTH, NUM_CHANNELS]`. Therefore, by providing `axis=[0, 1]`, we are asking TensorFlow to compute the average of all columns (`axis=0`) and all rows (`axis=1`), but not to average the RGB values (see Figure 2-4).

Printing out statistics of the image like this is helpful for another reason. If your input data is corrupt and there is unrepresentable floating-point data (technically called `NaN`) in your images, the mean will itself be `NaN`. This is a handy way to ensure that you haven't made a mistake when reading data.

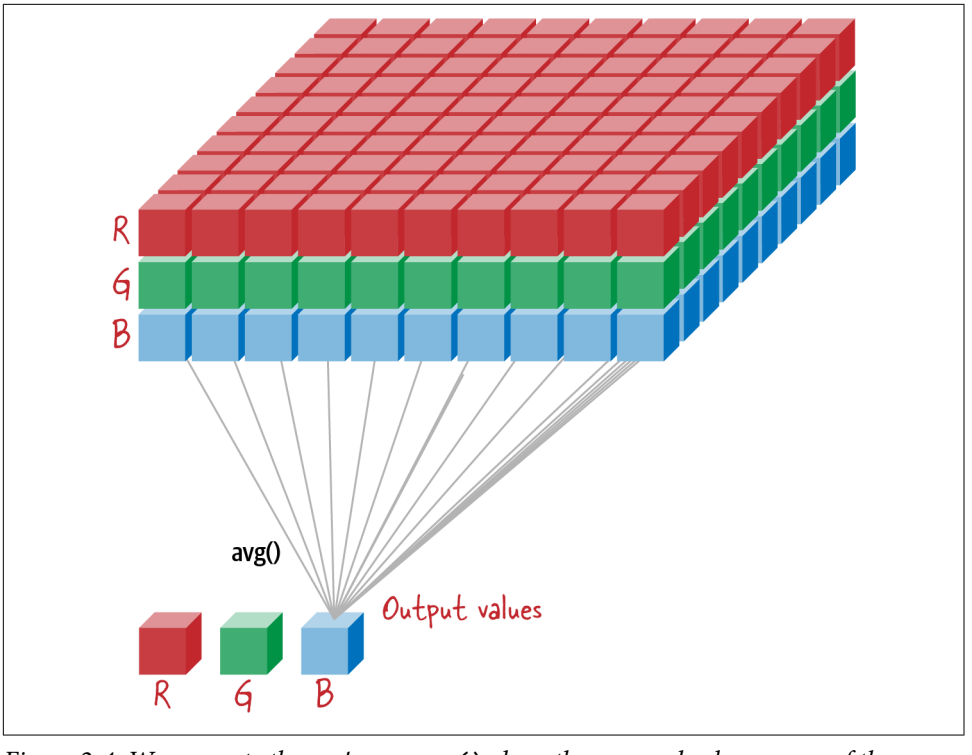


Figure 2-4. We compute the `reduce_mean()` along the row and column axes of the image.

A Linear Model Using Keras

As Figure 2-4 demonstrates, the `reduce_mean()` function weights each pixel value in the image the same. What if we were to apply a different weight to each of the width * height * 3 pixel-channel points in the image?

Given a new image, we can compute the weighted average of all its pixel values. We can then use this value to choose between the five types of flowers. Therefore, we will compute five such weighted averages (so that we are actually learning width * height * 3 * 5 weight values; see Figure 2-5), and choose the flower type based on which output is the largest.

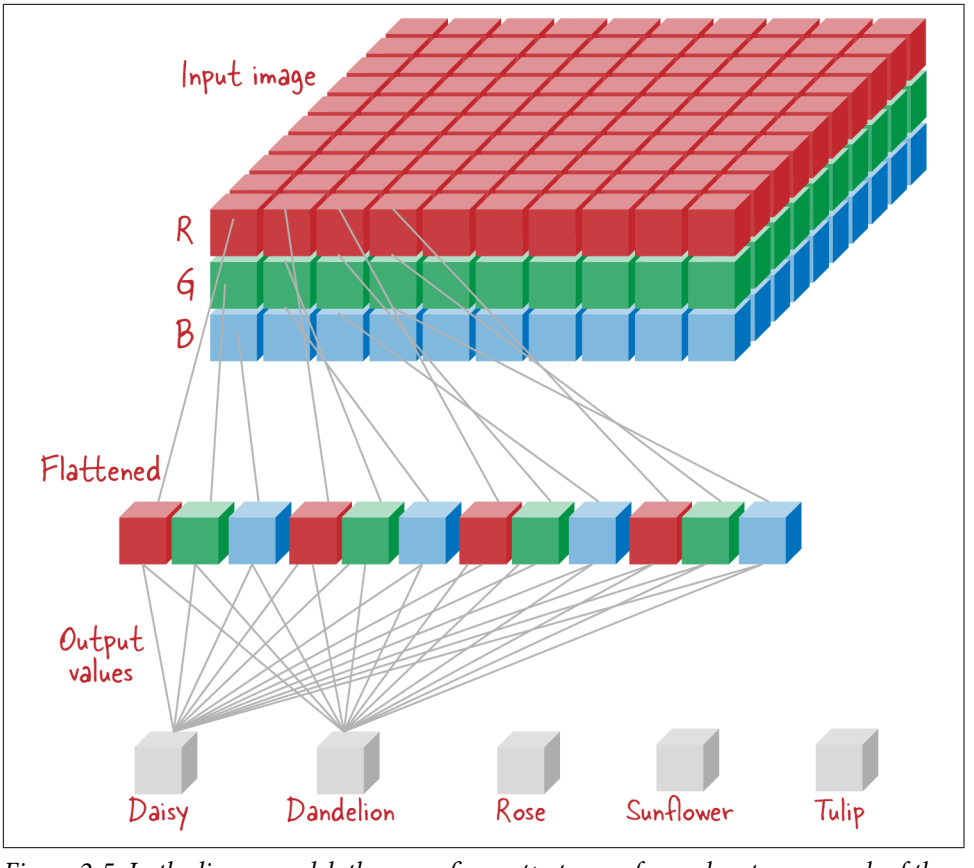


Figure 2-5. In the linear model, there are five outputs, one for each category; each of the output values is a weighted sum of the input pixel values.

In practice, a constant term called a *bias* is also added, so that we can represent each output value as:

$$Y_j = b_j + \sum_{rows} \sum_{columns} \sum_{channels} (w_i * x_i)$$

Without the bias, we'd be forcing the output to be zero if all the pixels are black.

Keras Model

Rather than write the preceding equation using low-level TensorFlow functions, it will be more convenient to use a higher-level abstraction. TensorFlow 1.1 shipped with one such abstraction, the Estimator API, and Estimators are still supported for

backward compatibility. However, the Keras API has been part of TensorFlow since TensorFlow 2.0, and it's what we recommend that you use.

A linear model can be represented in Keras as follows:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(IMG_HEIGHT, IMG_WIDTH, 3)),
    tf.keras.layers.Dense(len(CLASS_NAMES))
])
```

A *Sequential model* consists of *layers* that are connected such that the output of one layer is the input to the next. A layer is a Keras component that takes a tensor as input, applies some TensorFlow operations to that input, and outputs a tensor.

The first layer, which is implicit, is the input layer, which asks for a 3D image tensor. The second layer (the `Flatten` layer) takes a 3D image tensor as input and reshapes it to be a 1D tensor with the same number of values. The `Flatten` layer is connected to a `Dense` layer with one output node for each class of flower. The name `Dense` means that every output is a weighted sum of every input and no weights are shared. We will encounter other common types of layers later in this chapter.

To use the Keras model defined here, we need to call `model.fit()` with the training dataset and `model.predict()` with each image we want to classify. To train the model, we need to tell Keras how to optimize the weights based on the training dataset. The way to do this is to *compile* the model, specifying an *optimizer* to use, the *loss* to minimize, and *metrics* to report. For example:

```
model.compile(
    optimizer='adam',
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy'])
```

The Keras `predict()` function will do the model computation on the image. The parameters to the `compile()` function make more sense if we look at the prediction code first, so let's start there.

Prediction function

Because the model has the trained set of weights in its internal state, we can compute the predicted value for an image by calling `model.predict()` and passing in the image:

```
pred = model.predict(tf.reshape(img,
        [1, IMG_HEIGHT, IMG_WIDTH, NUM_CHANNELS]))
```

The reason for the `reshape()` is that `predict()` expects a batch of images, so we reshape the `img` tensor as a batch consisting of one image.

What does the `pred` tensor that is output from `model.predict()` look like? Recall that the final layer of the model was a `Dense` layer with five outputs, so the shape of `pred` is (5)—that is, it consists of five numbers corresponding to the five flower types. The first output is the model's confidence that the image in question is a daisy, the second output is the model's confidence that the image is a dandelion, and so on. The predicted confidence values are called *logits* and are in the range –infinity to +infinity.

The model's prediction is the label in which it has the highest confidence:

```
pred_label_index = tf.math.argmax(pred)
pred_label = CLASS_NAMES[pred_label_index]
```

We can convert the logits to probabilities by applying a function called the *softmax* function to them. So, the probability corresponding to the predicted label is:

```
prob = tf.math.softmax(pred)[pred_label_index]
```

Probability, Odds, Logits, Sigmoid, and Softmax

The output of a classification model is a *probability*—the likelihood that an event will occur over many trials. Therefore, when building classification models, it is important to understand several concepts related to probabilities. For example, if you are building a model that classifies a machine part as being defective or non-defective, the model does not provide a Boolean (true or false) output. Instead, it outputs the probability that the part is defective.

Suppose you have an event that can happen with a probability p. Then, the probability that it will *not* happen is $1 - p$. The *odds* that it will happen in any given trial is the probability of the event occurring divided by the probability of it not occurring, or $p / (1 - p)$. For example, if p=0.25, then the odds of the event happening are 0.25 / 0.75 = 1:3. On the other hand, if p=0.75, then the odds of it happening are 0.75 / 0.25 = 3:1.

The *logit* is the natural logarithm of the odds of the event happening. Thus, for an event with p=0.25, the logit is log(0.25 / 0.75), or –1.098. For an event with p=0.75, the logit is 1.098. As the probability approaches 0 the logit approaches –infinity, and as the probability approaches 1 the logit approaches +infinity. Therefore, the logit occupies the entire space of real numbers, as shown in Figure 2-6.

The *sigmoid* is the inverse of the logit function. Thus, the sigmoid of 1.098 is 0.75. Mathematically, the sigmoid is given by:

$$\sigma(Y) = \frac{1}{1 + e^{-Y}}$$

The sigmoid is in the range 0–1. If we have a `Dense` layer in Keras with one output node, by applying the sigmoid to it we can obtain a binary classifier that outputs a valid probability.

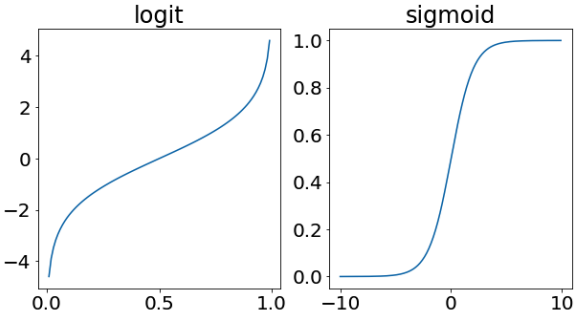


Figure 2-6. The logit and sigmoid functions are inverses of each other (the x-axis in the first graph and the y-axis in the second are the probability).

The *softmax* is the multiclass counterpart of the sigmoid. If you have N mutually exclusive events, and their logits are given by Y_j, then softmax(Y_j) provides the probability of the jth event. Mathematically, the softmax is given by:

$$S\left(Y_j\right) = \frac{e^{-Y_j}}{\sum_j e^{-Y_j}}$$

The softmax function is nonlinear and has the effect of squashing low values and boosting the maximum, as shown in Figure 2-7. Note that the sum of the probabilities in both instances adds up to 1.0. This property is useful because we can have a `Dense` layer in Keras with five output nodes, and by applying the softmax to it, we can obtain a sound probability distribution.

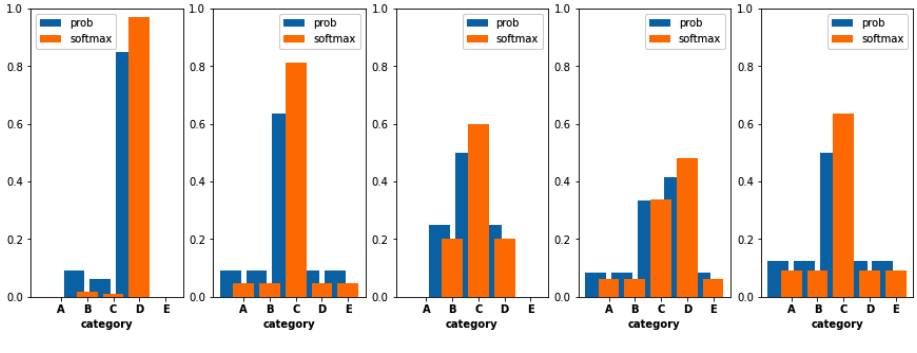


Figure 2-7. The softmax function squashes low values and boosts the maximum value.

Activation function

It is not sufficient to simply call `model.predict()`, because `model.predict()` returns a weighted sum that is unbounded. We can treat this weighted sum as logits and apply either the sigmoid or the softmax function (depending on whether we have a binary classification problem or a multiclass one) to obtain the probability:

```
pred = model.predict(tf.reshape(img,
                    [1, IMG_HEIGHT, IMG_WIDTH, NUM_CHANNELS]))
prob = tf.math.softmax(pred)[pred_label_index]
```

We can make things more convenient for end users if we add an *activation function* to the last layer of the model:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(IMG_HEIGHT, IMG_WIDTH, 3)),
    tf.keras.layers.Dense(len(CLASS_NAMES), activation='softmax')
])
```

If we do this, then `model.predict()` will return five probabilities (not logits), one for each class. There is no need for the client code to call `softmax()`.

Any layer in Keras can have an activation function applied to its output. Supported activation functions include `linear`, `sigmoid`, and `softmax`. We'll look at other activation functions later in this chapter.

Optimizer

Keras allows us to choose the optimizer that we wish to use to tune the weights based on the training dataset. Available optimizers include:

Stochastic gradient descent (SGD)
: The most basic optimizer.

Adagrad (adaptive gradients) and Adam
: Improve upon the basic optimizer by adding features that allow for faster convergence.

Ftrl
: An optimizer that tends to work well on extremely sparse datasets with many categorical features.

Adam is the tried-and-proven choice for deep learning models. We recommend using Adam as your optimizer for computer vision problems unless you have a strong reason not to.

SGD and all its variants, including Adam, rely on receiving mini-batches (often just called *batches*) of data. For each batch of data, we feed forward through the model and calculate the error and the *gradient*, or how much each weight contributed to the error; then the optimizer updates the weights with this information, ready for the

next batch of data. Therefore, when we read the training dataset, we have to also batch it:

```
train_dataset = (tf.data.TextLineDataset(
    "gs://cloud-ml-data/img/flower_photos/train_set.csv").
    map(decode_csv)).batch(10)
```

Gradient Descent

Training the neural network actually means using training images and labels to adjust weights and biases so as to minimize the cross-entropy loss. The cross-entropy is a function of the model's weights and biases, the pixels of the training image, and its known class.

If we compute the extent to which the cross-entropy changes when we adjust each of the weights independently, we get the *partial derivative* of the cross-entropy. We can compute the gradient in different directions from the partial derivative computed for a given image, label, and current weights and biases. The mathematical property of a gradient is that it points "up" if, by moving in that direction, the loss increases. Since we want to go where the cross-entropy is low, we go in the direction where the gradient decreases the most. To do this, we update the weights and biases by a fraction of the gradient. We then do the same thing again and again using the next batches of training images and labels, in a training loop. The training process is depicted in Figure 2-8. The hope is that this will converge to a place where the cross-entropy is minimal, although nothing guarantees that this minimum is unique or even that it is the global minimum.

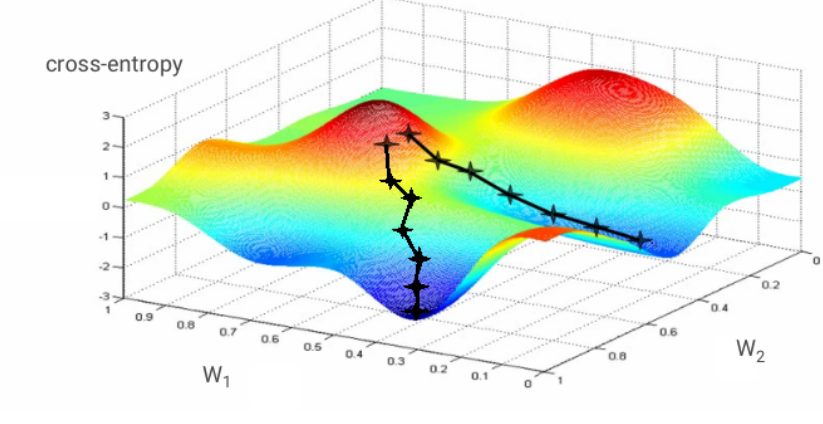


Figure 2-8. The training process involves taking small steps in the direction where the loss decreases the most.

Note that you cannot update your weights and biases by the whole length of the gradient at each iteration—you would be jumping from one side of the valley to the

other. To get to the bottom, you need to do smaller steps by using only a fraction of the gradient, typically in the 1/1,000 range. This fraction is called the learning rate; we'll discuss it in more detail later in this chapter.

You can compute your gradient on just one example image and update the weights and biases immediately, but doing so on a batch of, say, 128 images gives a gradient that better represents the constraints imposed by different example images and is therefore likely to converge toward the solution faster. The size of the mini-batch is an adjustable parameter.

This technique is called *stochastic gradient descent*. It has another, more pragmatic benefit: working with batches also means working with larger matrices, and these are usually easier to optimize on GPUs and TPUs (tensor processing units, which are specialized hardware to accelerate machine learning operations).

Training loss

The optimizer tries to choose the weights that minimize the model's error on the training dataset. For classification problems, there are strong mathematical reasons to choose cross-entropy as the error to be minimized. To calculate the cross-entropy, we compare the output probability (p_j for the jth class) of the model against the true label for that class (L_j) and sum this up over all the classes using the formula:

$$\sum_j -L_j \log\left(p_j\right)$$

In other words, we take the logarithm of the probability for predicting the correct label. If the model gets it exactly correct, this probability will be 1; log(1) is 0, and so the loss is 0. If the model gets it exactly wrong, this probability will be 0; log(0) is –infinity, and so the loss is +infinity, the worst possible loss. Using cross-entropy as our error measure allows us to tune the weights based on small improvements in the probability assigned to the correct label.

In order to compute the loss, the optimizer will need to compare the label (returned by the `parse_csvline()` function) with the output of `model.predict()`. The specific loss you use will depend on how you are representing the label and what the last layer of your model returns.

If your labels are one-hot encoded (e.g., if the label is encoded as [1 0 0 0 0] for daisy images), then, you should use *categorical cross-entropy* as your loss function. This will show up in your `decode_csv()` as follows:

```
def parse_csvline(csv_row):
    record_defaults = ["path", "flower"]
    filename, label_string = tf.io.decode_csv(csv_row, record_defaults)
    img = read_and_decode(filename, [IMG_HEIGHT, IMG_WIDTH])
```

```
    label = tf.math.equal(CLASS_NAMES, label_string)
    return img, label
```

Because `CLASS_NAMES` is an array of strings, comparing to a single label will return a one-hot-encoded array where the Boolean value is 1 in the corresponding position. You will specify the loss as follows:

```
tf.keras.losses.CategoricalCrossentropy(from_logits=False)
```

Note that the constructor takes a parameter which specifies whether the last layer of the model returns logits of probabilities, or whether you have done a softmax.

On the other hand, if your labels will be represented as integer indices (e.g., 4 indicates tulips), then your `decode_csv()` will represent the label by the position of the correct class:

```
label = tf.argmax(tf.math.equal(CLASS_NAMES, label_string))
```

And the loss will be specified as:

```
tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False)
```

Again, take care to specify the value of `from_logits` appropriately.

Why Have Two Ways to Represent the Label?

When we do one-hot encoding, we represent a flower that is a daisy as [1 0 0 0 0] and a flower that is a tulip as [0 0 0 0 1]. The length of the one-hot-encoded vector is the number of classes. The sparse representation would be 0 for daisy and 4 for tulip. The sparse representation takes up less space (especially if there are hundreds or thousands of possible classes) and is therefore much more efficient.

Why, then, does Keras support two ways to represent labels?

The sparse representation will not work if the problem is a multilabel multiclass problem. If an image can contain both daisies and tulips, it is quite straightforward to encode this using the one-hot-encoded representation: [1 0 0 0 1]. With the sparse representation, there is no way to represent this scenario unless you are willing to create separate categories for each possible combination of categories.

Therefore, we recommend you use the sparse representation for most problems; but remember that one-hot encoding the labels and using the `CategoricalCrossentropy()` loss function will help you handle multilabel multiclass situations.

Error metrics

While we can use the cross-entropy loss to minimize the error on the training dataset, business users will typically want a more understandable error metric. The most

common error metric that is used for this purpose is *accuracy*, which is simply the fraction of instances that are classified correctly.

However, the accuracy metric fails when one of the classes is very rare. Suppose you are trying to identify fake ID cards, and your model has the following performance characteristics:

	Card identified as fake	Card identified as genuine
Actual fake ID cards	8 (TP)	2 (FN)
Actual genuine ID cards	140 (FP)	850 (TN)

The dataset has 990 genuine ID cards and 10 fake ID cards—there is a *class imbalance*. Of the fake ID cards, 8 were correctly identified as fake. These are the true positives (TP). The accuracy on this dataset would thus be (850 + 8) / 1,000, or 0.858. It can be immediately seen that because fake ID cards are so rare, the model's performance on this class has very little impact on its overall accuracy score—even if the model correctly identified only 2 of the 10 fake ID cards, the accuracy would remain nearly the same: 0.852. Indeed, the model can achieve an accuracy of 0.99 simply by identifying all cards as being valid! In such cases, it is common to report two other metrics:

Precision
: The fraction of true positives in the set of identified positives: TP / (TP + FP). Here, the model has identified 8 true positives and 140 false positives, so the precision is only 8/148. This model is very imprecise.

Recall
: The fraction of true positives identified among all the positives in the dataset: TP / (TP + FN). Here, there are 10 positives in the full dataset and the model has identified 8 of them, so the recall is 0.8.

In addition to the precision and recall, it is also common to report the F1 score, which is the harmonic mean of the two numbers:

$$F1 = 2/\left[\frac{1}{precision} + \frac{1}{recall}\right]$$

In a binary classification problem such as the one we're considering here (identifying fake ID cards), the accuracy, precision, and recall all rely on the probability threshold we choose to determine whether to classify an instance in one category or the other. By varying the probability threshold, we can obtain different trade-offs in terms of precision and recall. The resulting curve is called the *precision-recall curve* (see Figure 2-9). Another variant of this curve, where the true positive rate is plotted against the false positive rate, is called the *receiver operating characteristic* (ROC)

curve. The area under the ROC curve (commonly shortened as *AUC*) is also often used as an aggregate measure of performance.

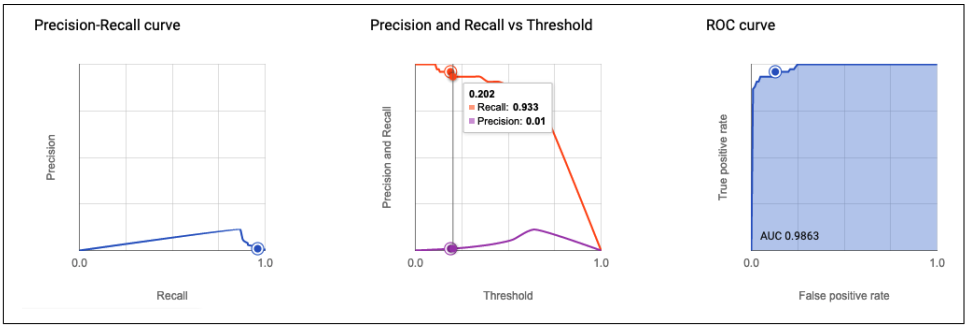


Figure 2-9. By varying the threshold, it is possible to get different precision and recall measures.

We normally want to report these metrics not on the training dataset, but on an independent evaluation dataset. This is to verify that the model hasn't simply memorized the answers for the training dataset.

Training the Model

Let's now put all the concepts that we covered in the previous section together to create and train a Keras model.

Creating the datasets

To train a linear model, we need a training dataset. Actually, we want two datasets—a training dataset and an evaluation dataset—to verify whether the trained model *generalizes*, or works on data that it has not seen during training.

So, we first obtain the training and evaluation datasets:

```
train_dataset = (tf.data.TextLineDataset(
    "gs://cloud-ml-data/img/flower_photos/train_set.csv").
    map(decode_csv)).batch(10)

eval_dataset = (tf.data.TextLineDataset(
    "gs://cloud-ml-data/img/flower_photos/eval_set.csv").
    map(decode_csv)).batch(10)
```

where `decode_csv()` reads and decodes JPEG images:

```
def decode_csv(csv_row):
    record_defaults = ["path", "flower"]
    filename, label_string = tf.io.decode_csv(csv_row, record_defaults)
    img = read_and_decode(filename, [IMG_HEIGHT, IMG_WIDTH])
    label = tf.argmax(tf.math.equal(CLASS_NAMES, label_string))
    return img, label
```

The `label` that is returned in this code is the sparse representation—i.e., the number 4 for tulips, that class's index—and not the one-hot-encoded one. We batch the training dataset because the optimizer class expects batches. We also batch the evaluation dataset to avoid creating two versions of all our methods (one that operates on batches, and another that requires one image at a time).

Creating and viewing the model

Now that the datasets have been created, we need to create the Keras model that is to be trained using those datasets:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(IMG_HEIGHT, IMG_WIDTH, 3)),
    tf.keras.layers.Dense(len(CLASS_NAMES), activation='softmax')
])
model.compile(optimizer='adam',
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=['accuracy'])
```

We can view the model using:

```
tf.keras.utils.plot_model(model, show_shapes=True, show_layer_names=False)
```

This yields the diagram in Figure 2-10. Note that the input layer takes a batch (that's the ?) of [224, 224, 3] images. The question mark indicates that the size of this dimension is undefined until runtime; this way, the model can dynamically adapt to any batch size. The `Flatten` layer takes this input and returns a batch of 224 * 224 * 3 = 150,528 numbers that are then connected to five outputs in the `Dense` layer.

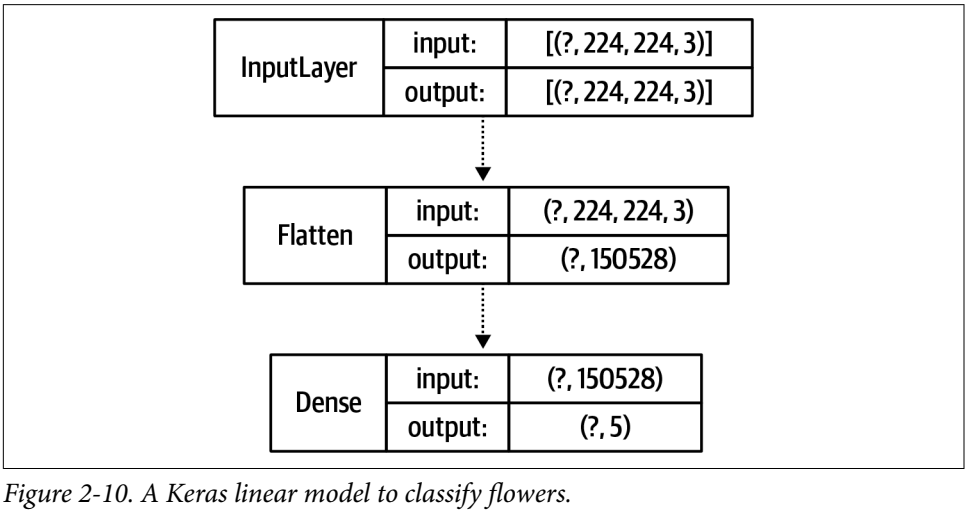


Figure 2-10. A Keras linear model to classify flowers.

We can verify that the `Flatten` operation does not need any trainable weights, but the `Dense` layer has 150,528 * 5 = 752,645 weights that need to be trained by using `model.summary()`, which yields:

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 150528)            0
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 752645
=================================================================
Total params: 752,645
Trainable params: 752,645
Non-trainable params: 0
```

Fitting the model

Next, we train the model using `model.fit()` and pass in the training and validation datasets:

```
history = model.fit(train_dataset,
                    validation_data=eval_dataset, epochs=10)
```

Note that we are passing in the training dataset to train on, and the validation dataset to report accuracy metrics on. We are asking the optimizer to go through the training data 10 times (an *epoch* is a full pass through the dataset). We hope that 10 epochs will be sufficient for the loss to converge, but we should verify this by plotting the history of the loss and error metrics. We can do that by looking at what the history has been tracking:

```
history.history.keys()
```

We obtain the following list:

```
['loss', 'accuracy', 'val_loss', 'val_accuracy']
```

We can then plot the loss and validation loss using:

```
plt.plot(history.history['val_loss'], ls='dashed');
```

This yields the graph shown in the lefthand panel of Figure 2-11. Note that the loss does not go down smoothly; instead, it is quite choppy. This is an indication that our choices of batch size and optimizer settings can be improved—unfortunately, this part of the ML process is trial and error. The validation loss goes down, and then starts to increase. This is an indication that *overfitting* is starting to happen: the network has started to memorize details of the training dataset (such details are called *noise*) that do not occur in the validation dataset. Either 10 epochs is too long, or we need to add regularization. Overfitting and regularization are topics that we will address in more detail in the next section.

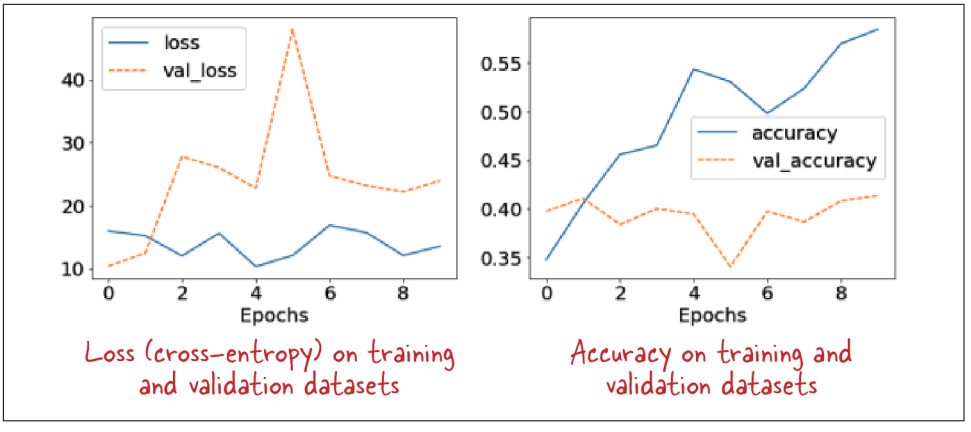


Figure 2-11. Loss and accuracy curves on the training (solid) and validation (dashed) sets.

It is also possible to plot the accuracy on the training dataset and the validation dataset using:

```
training_plot('accuracy', history)
```

The resulting graph is shown in the righthand panel of Figure 2-11. Notice that the accuracy on the training dataset goes on increasing the longer we train, while the accuracy on the validation dataset plateaus.

These lines are also choppy, providing us with the same insights we got from the loss curves. However, the accuracy that we have obtained (0.4) on the evaluation dataset is better than what we would have gotten from random chance (0.2). This indicates the model has been able to learn and become somewhat skillful at the task.

Plotting predictions

We can look at what the model has learned by plotting its predictions on a few images from the training dataset:

```
batch_image = tf.reshape(img, [1, IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS])
batch_pred = model.predict(batch_image)
pred = batch_pred[0]
```

Note that we need to take the single image that we have and make it a batch, because that's what the model was trained on and what it expects. Fortunately, we don't need to pass in exactly 10 images (our batch size was 10 during training) because the model was designed to take any batch size (recall that the first dimension in Figure 2-10 was a ?).

The first few predictions from the training and evaluation datasets are shown in Figure 2-12.

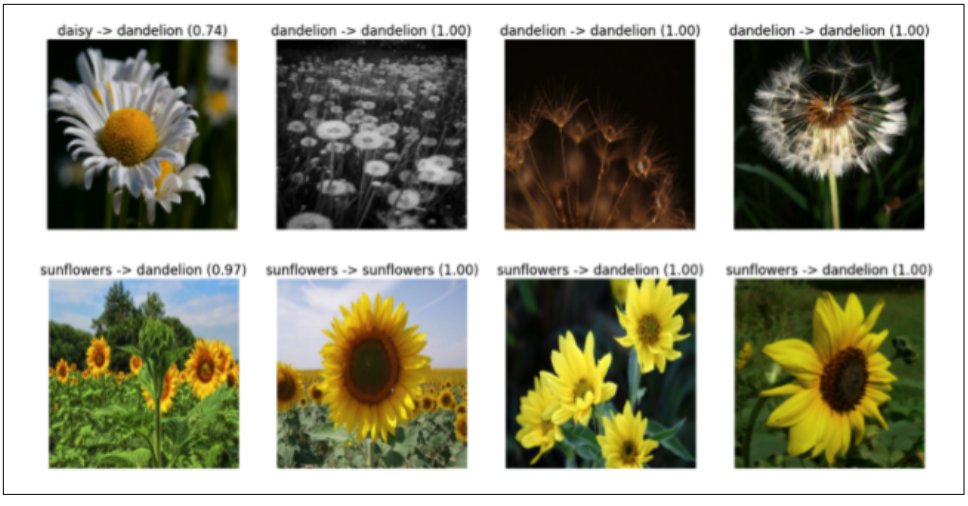


Figure 2-12. The first few images from the training (top row) and evaluation (bottom row) datasets—the first image, which is actually a daisy, has been wrongly classified as a dandelion with a probability of 0.74.

Image Regression

So far we've been focusing on the task of image classification. Another computer vision problem that you might encounter, though it's far less common, is image *regression*. One reason we might want to do this is because we want to measure something within the image. The problem here isn't counting the number of a certain type of object, which is solved another way that we will discuss in Chapter 11, but instead measuring a more real-valued property like height, length, volume, etc.

For instance, we may want to predict the rainfall amount from aerial images of cloud cover over the region of interest. By training an image regression model with tiles of the cloud images as input (see Figure 2-13, where two tiles are marked) and the precipitation amounts on the ground as our labels, we'd be able to learn the mapping from cloud images to precipitation.

Since we are measuring precipitation amounts in millimeters (mm), our labels are continuous, real numbers. We could of course reframe this as a classification problem by bucketing certain amounts of precipitation into categories such as low, medium, and high, but that may not be relevant for this specific use case.

Fortunately, regression isn't more complicated than image classification. We merely have to change our final neural network layer, the `Dense` output layer, from having a `sigmoid` or `softmax` activation to `None`, and change the number of `units` to the number of regression predictions we want to make from this one image (in this hypothetical case, just one).

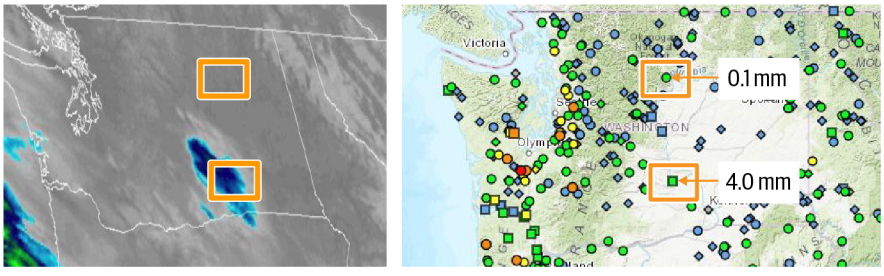


Figure 2-13. Image regression to learn to predict rainfall amount—tiles of the cloud cover image on the left are treated as input, and the labels are measurements of precipitation on the ground (measured by a rain gauge at the center of the tile). Images courtesy of NOAA (left) and USGS (right).

The code would look like this:

```
tf.keras.layers.Dense(units=1, activation=None)
```

In addition, since this is now a regression problem, we should use a regression loss function, such as mean squared error (MSE):

```
tf.keras.losses.MeanSquaredError()
```

Once our model is trained, at inference time we can provide the model with images of clouds and it will return predictions for the amount of precipitation on the ground.

A Neural Network Using Keras

In the linear model that we covered in the previous section, we wrote the Keras model as:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(IMG_HEIGHT, IMG_WIDTH, 3)),
    tf.keras.layers.Dense(len(CLASS_NAMES), activation='softmax')
])
```

The output is the softmax of the weighted average of the flattened input pixel values:

$$Y = \text{softmax}\left(B + \sum_{pixels} W_i X_i\right)$$

B is the bias tensor, W the weights tensor, X the input tensor, and Y the output tensor. This is usually written in matrix form as (using § to represent the softmax):

$$Y = §(B + WX)$$

As shown in Figure 2-10, and in the following model summary, there is only one trainable layer, the `Dense` one. The `Flatten` operation is a reshaping operation and does not contain any trainable weights:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 150528)            0
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 752645
=================================================================
```

Linear models are great, but they are limited in what they can model. How do we obtain more complex models?

Neural Networks

One way to get a more complex model is to interpose one or more `Dense` layers in between the input and output layers. This results in a machine learning model called a *neural network*, for reasons that we will explain shortly.

Hidden layers

Suppose that we interpose one more `Dense` layer using:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(IMG_HEIGHT, IMG_WIDTH, 3)),
    tf.keras.layers.Dense(128),
    tf.keras.layers.Dense(len(CLASS_NAMES), activation='softmax')
])
```

The model now has three layers (see Figure 2-14). A layer with trainable weights, such as the one we added that is neither the input nor the output layer, is called a *hidden* layer.

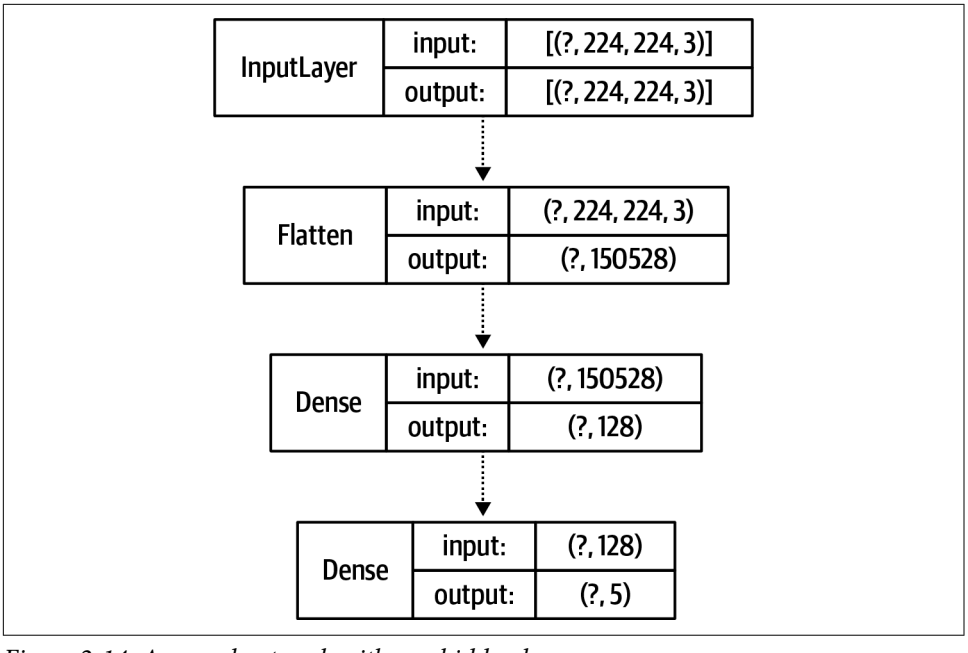


Figure 2-14. A neural network with one hidden layer.

Mathematically, the output is now:

$$Y = \S(B_2 + W_2(B_1 + W_1X))$$

Simply wrapping multiple layers like this is pointless, since we might as well have multiplied the second layer's weight (W_2) into the equation—the model remains a linear model. However, if we add a nonlinear *activation* function $A(x)$ to transform the output of the hidden layer:

$$Y = \S(B_2 + W_2A(B_1 + W_1X))$$

then the output becomes capable of representing more complex relationships than a simple linear function.

In Keras, we introduce the activation function as follows:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(IMG_HEIGHT, IMG_WIDTH, 3)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(len(CLASS_NAMES), activation='softmax')
])
```

The rectified linear unit (ReLU) is the most commonly used activation function for hidden layers (see Figure 2-15). Other commonly used activation functions include *sigmoid*, *tanh*, and *elu*.

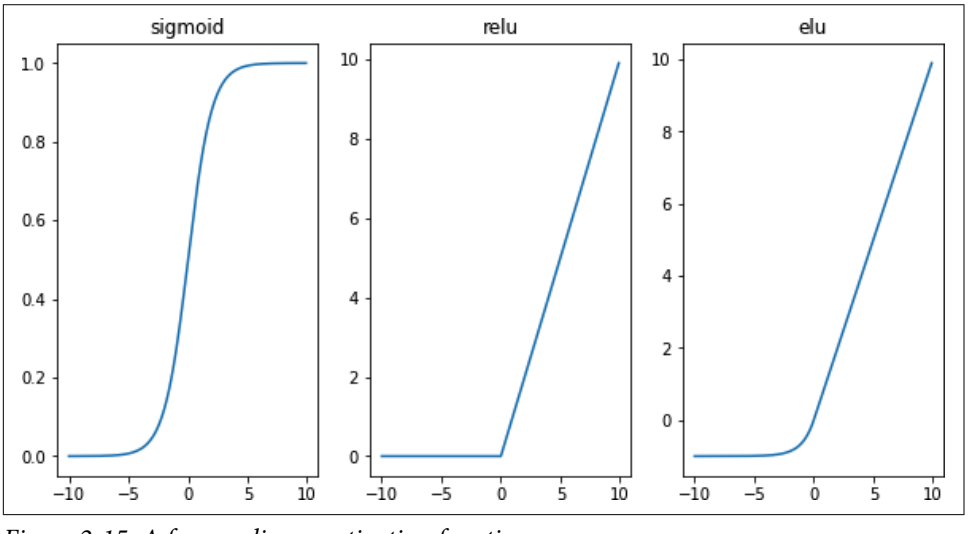


Figure 2-15. A few nonlinear activation functions.

All three of the activation functions shown in Figure 2-15 are loosely based on how neurons in the human brain fire if the input from the dendrites together exceeds some minimum threshold (see Figure 2-16). Thus, a model that has a hidden layer with a nonlinear activation function is called a "neural network."

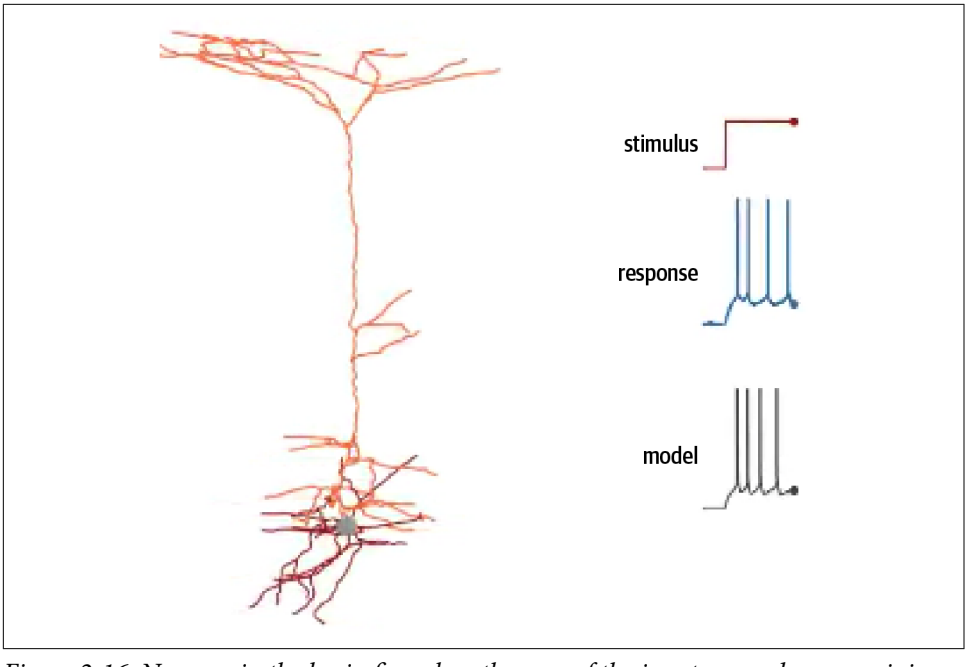


Figure 2-16. Neurons in the brain fire when the sum of the inputs exceeds some minimum threshold. Image credit: Allen Institute for Brain Science, Allen Human Brain Atlas, available from human.brain-map.org.

The sigmoid is a continuous function that behaves most similarly to how brain neurons work—the output saturates at both extremes. However, the sigmoid function suffers from slow convergence because the weight update at each step is proportional to the gradient, and the gradient near the extremes is very small. The ReLU is more often used so that the weight updates remain the same size in the active part of the function. In a `Dense` layer with a ReLU activation function, the activation function "fires" if the weighted sum of the inputs is greater than $-b$, where b is the bias. The strength of the firing is proportional to the weighted sum of the inputs. The issue with a ReLU is that it is zero for half its domain. This leads to a problem called *dead ReLUs*, where no weight update ever happens. The elu activation function (see Figure 2-15) solves this problem by having a small exponential negative value instead of zero. It is, however, expensive to compute because of the exponential. Therefore, some ML practitioners instead use the Leaky ReLU, which uses a small negative slope.

Training the neural network

Training the neural network is similar to training the linear model. We compile the model, passing in the optimizer, the loss, and the metrics. Then we call `model.fit()`, passing in the datasets:

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(
                  from_logits=False),
              metrics=['accuracy'])
history = model.fit(train_dataset,
                    validation_data=eval_dataset,
                    epochs=10)
```

The result, shown in Figure 2-17, reveals that the best validation accuracy that we have obtained (0.45) is similar to what we obtained with a linear model. The curves are also not smooth.

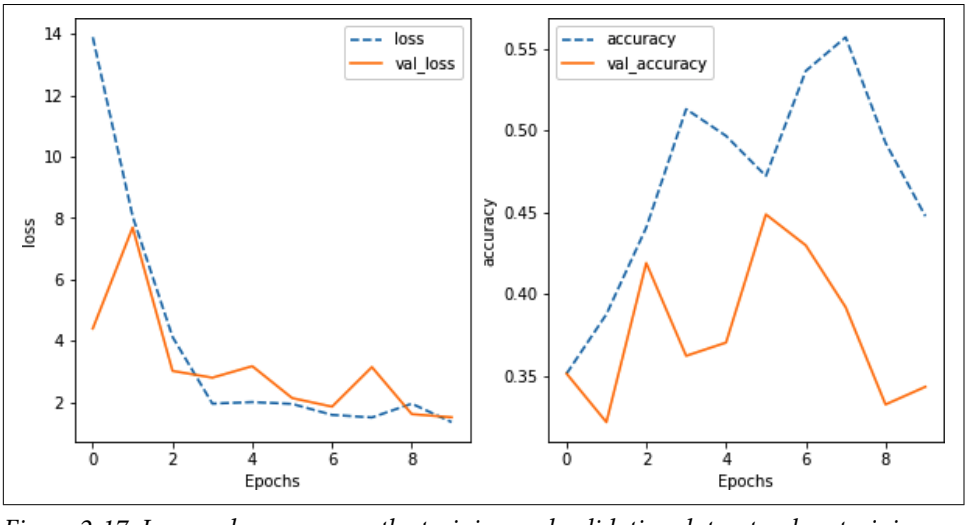


Figure 2-17. Loss and accuracy on the training and validation datasets when training a neural network.

We would normally expect that adding layers to a model will improve the ability of the model to fit the training data, and thus lower the loss. That is, indeed, the case—whereas the cross-entropy loss for the linear model is on the order of 10, it is on the order of 2 for the neural network. However, the accuracies are pretty similar, indicating that much of the improvement is obtained by the model driving probabilities like 0.7 to be closer to 1.0 than by getting items misclassified by the linear model correct.

There are still some improvements that we can try, though. For example, we can change the learning rate and the loss function, and make better use of the validation dataset. We'll look at these next.

Learning rate

A gradient descent optimizer works by looking in all directions at each point and picking the direction where the error function is decreasing the most rapidly. Then it makes a step in that direction and tries again. For example, in Figure 2-18, starting at

the first point (the circle marked 1), the optimizer looks in two directions (actually 2^N directions, where N is the dimension of the weight tensor to be optimized) and chooses direction 2, because it is the direction in which the loss function decreases the fastest. Then, the optimizer updates the weight value by making a *step* in that direction, as indicated by the dashed curved line. The size of this step for every weight value is proportional to a model hyperparameter called the *learning rate*.

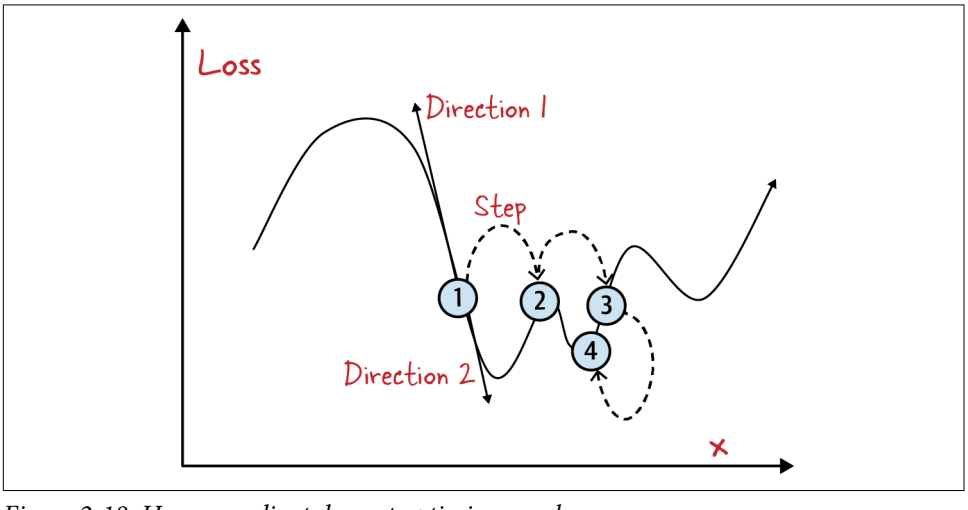


Figure 2-18. How a gradient descent optimizer works.

As you can see, if the learning rate is too high, the optimizer might skip over the minima completely. After this step (denoted by the circle marked 2 in the figure), the optimizer again looks in two directions and then continues to the third point, because the loss curve is dropping faster in that direction. After this step, the gradient is evaluated again. Now the direction points backward, and the optimizer manages to find the local minimum in between the second and third points. The global minimum which was between the first and second steps has, however, been missed.

In order to not skip over minima, we should use a small value for the learning rate. But if the learning rate is too small, the model will get stuck in a local minimum. Also, the smaller the value of the learning rate, the slower the model will converge. Thus, there's a trade-off between not missing minima and getting the model to converge quickly.

The default learning rate for the Adam optimizer is 0.001. We can change it by changing the optimizer passed into the `compile()` function:

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
              loss=..., metrics=...)
```

Repeating the training with this lower training rate, we get the same end result in terms of accuracy, but the curves are noticeably less choppy (see Figure 2-19).

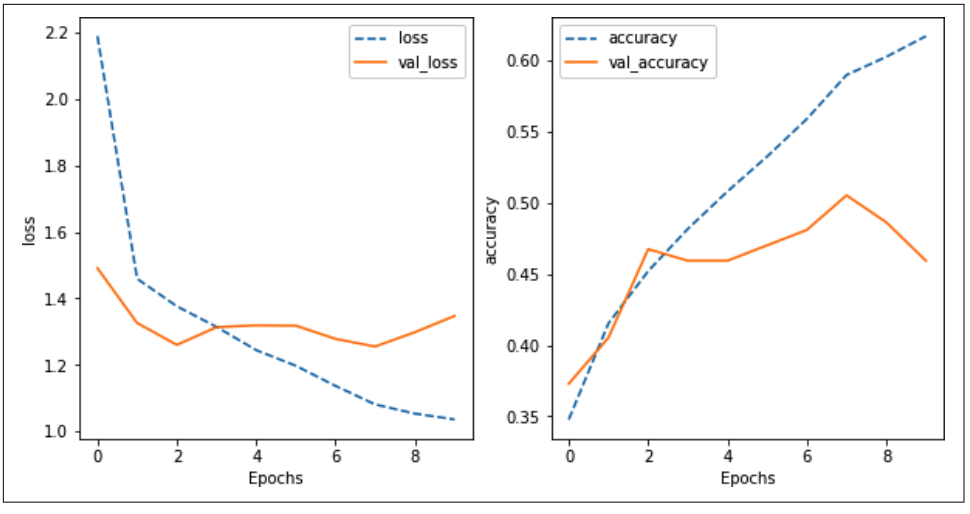


Figure 2-19. The loss and accuracy curves when the learning rate is lowered to 0.0001.

Regularization

It is also worth noting that the number of trainable parameters in the neural network is 128 times the number of trainable parameters in the linear model (19 million versus 750,000). Yet, we have only about 3,700 images. The more complex model might perform better, but we'd probably need much more data—on the order of hundreds of thousands of images. Later in this book, we will look at data augmentation techniques to make the most out of the data that we do have.

Given that we have a relatively small dataset for the complexity of the model that we are using, it is possible that the model will start to use individual trainable weights to "memorize" the classification answers for individual images in the training dataset—this is the overfitting that we can see happening in Figure 2-19 (the loss on the validation set started to increase even though the training accuracy was still decreasing). When this happens, the weight values start to become highly tuned to very specific pixel values and attain very high values.[1] Therefore, we can reduce the incidence of overfitting by changing the loss to apply a penalty on the weight values themselves. This sort of penalty applied to the loss function is called *regularization*.

Two common forms are:

$$loss = cross{-}entropy + \sum_i |w_i|$$

1 A good non-mathematical explanation of this phenomenon can be found at DataCamp.com.

and:

$$loss = cross{-}entropy + \sum_i w_i^2$$

The first type of penalty is called an *L1 regularization term*, and the second is called an *L2 regularization term*. Either penalty will cause the optimizer to prefer smaller weight values. L1 regularization drives many of the weight values to zero but is more tolerant of individual large weight values than L2 regularization, which tends to drive all the weights to small but nonzero values. The mathematical reasons why this is the case are beyond the scope of this book, but it's useful to understand that we use L1 if we want a compact model (because we can prune zero weights), whereas we use L2 if we want to limit overfitting to the maximum possible.

This is how we apply a regularization term to the `Dense` layers:

```
regularizer = tf.keras.regularizers.l1_l2(0, 0.001)
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(
                          IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS)),
    tf.keras.layers.Dense(num_hidden,
                          kernel_regularizer=regularizer,
                          activation=tf.keras.activations.relu),
    tf.keras.layers.Dense(len(CLASS_NAMES),
                          kernel_regularizer=regularizer,
                          activation='softmax')
])
```

With L2 regularization turned on, we see from Figure 2-20 that the loss values are higher (because they include the penalty term). However, it is clear that overfitting is still happening after epoch 6. This indicates that we need to increase the regularization amount. Again, this is trial and error.

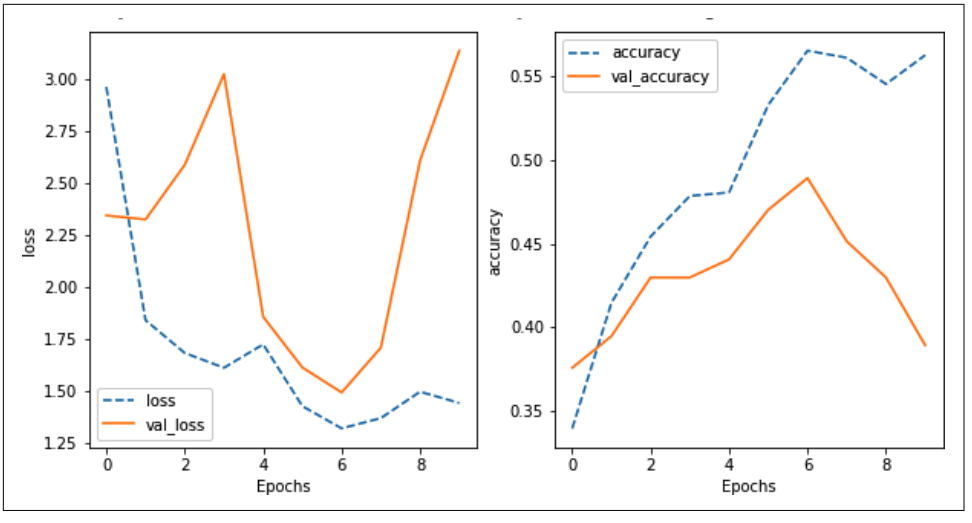


Figure 2-20. The loss and accuracy curves when L2 regularization is added.

Early stopping

Look carefully at the righthand panel in Figure 2-20. Both the training and validation set accuracies increase smoothly until the sixth epoch. After that, even though the training set accuracy continues to increase, the validation accuracy starts to drop. This is a classic sign that the model has stopped generalizing to unseen data, and is now starting to fit noise in the training dataset.

It would be good if we could stop the training once the validation accuracy stops increasing. In order to do that, we pass in a callback to the `model.fit()` function:

```
history = model.fit(train_dataset,
    validation_data=eval_dataset,
    epochs=10,
    callbacks=[tf.keras.callbacks.EarlyStopping(patience=1)]
)
```

Because convergence can be a bit bumpy, the `patience` parameter allows us to configure the number of epochs for which we want the validation accuracy to not decrease before training is stopped.

Add in the `EarlyStopping()` callback only after you have tuned the learning rate and regularization to get smooth, well-behaved training curves. If your training curves are choppy, it is possible that you will miss out on obtaining better performance by stopping early.

Hyperparameter tuning

We chose a number of parameters for our model: the number of hidden nodes, the learning rate, the L2 regularization, and so on. How do we know that these are optimal? We don't. We need to *tune* these hyperparameters.

One way to do this is to use the Keras Tuner. To use the Keras Tuner, we implement the model-building function to use hyperparameters (the full code is in a *02_ml_models/02b_neural_network.ipynb* on GitHub):

```
import kerastuner as kt

# parameterize to the values in the previous cell
def build_model(hp):
    lrate = hp.Float('lrate', 1e-4, 1e-1, sampling='log')
    l1 = 0
    l2 = hp.Choice('l2', values=[0.0, 1e-1, 1e-2, 1e-3, 1e-4])
    num_hidden = hp.Int('num_hidden', 32, 256, 32)

    regularizer = tf.keras.regularizers.l1_l2(l1, l2)

    # NN with one hidden layer
    model = tf.keras.Sequential([
        tf.keras.layers.Flatten(
            input_shape=(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS)),
        tf.keras.layers.Dense(num_hidden,
                              kernel_regularizer=regularizer,
                              activation=tf.keras.activations.relu),
        tf.keras.layers.Dense(len(CLASS_NAMES),
                              kernel_regularizer=regularizer,
                              activation='softmax')
    ])
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=lrate),
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(
                              from_logits=False),
                  metrics=['accuracy'])
  return model
```

As you can see, we defined the space from which the hyperparameters are drawn. The learning rate (`lrate`) is a floating-point value between 1e-4 and 1e-1, chosen logarithmically (not linearly). The L2 regularization value is chosen from a set of five predefined values (0.0, 1e-1, 1e-2, 1e-3, and 1e-4). The number of hidden nodes (`num_hidden`) is an integer chosen from the range 32 to 256 in increments of 32. These values are then used in the model-building code as normal.

We pass the `build_model()` function into a Keras Tuner optimization algorithm. Several algorithms are supported, but Bayesian optimization is an old standby that works well for computer vision problems:

```
tuner = kt.BayesianOptimization(
    build_model,
    objective=kt.Objective('val_accuracy', 'max'),
    max_trials=10,
    num_initial_points=2,
    overwrite=False) # True to start afresh
```

Here, we are specifying that our objective is to maximize the validation accuracy and that we want the Bayesian optimizer to run 10 trials starting from 2 randomly chosen seed points. The tuner can pick up where it left off, and we are asking Keras to do so by telling it to reuse information learned in preexisting trials and not start with a blank slate.

Having created the tuner, we can then run the search:

```
tuner.search(
    train_dataset, validation_data=eval_dataset,
    epochs=5,
    callbacks=[tf.keras.callbacks.EarlyStopping(patience=1)]
)
```

At the end of the run, we can get the top *N* trials (the ones that ended with the highest validation accuracy) using:

```
topN = 2
for x in range(topN):
    print(tuner.get_best_hyperparameters(topN)[x].values)
    print(tuner.get_best_models(topN)[x].summary())
```

When we did hyperparameter tuning for the 5-flowers problem, we determined that the best set of parameters was:

```
{'lrate': 0.00017013245197465996, 'l2': 0.0, 'num_hidden': 64}
```

The best validation accuracy obtained was 0.46.

Deep Neural Networks

The linear model gave us an accuracy of 0.4. The neural network with one hidden layer gave us an accuracy of 0.46. What if we add more hidden layers?

A *deep neural network* (DNN) is a neural network with more than one hidden layer. Each time we add a layer, the number of trainable parameters increases. Therefore, we will need a larger dataset. We still have only 3,700 flower images, but, as you'll see, there are a few tricks (namely dropout and batch normalization) that we can use to limit the amount of overfitting that happens.

Building a DNN

We can parameterize the creation of a DNN as follows:

```
def train_and_evaluate(batch_size = 32,
                       lrate = 0.0001,
                       l1 = 0,
                       l2 = 0.001,
                       num_hidden = [64, 16]):
    ...

    # NN with multiple hidden layers
    layers = [
              tf.keras.layers.Flatten(
                  input_shape=(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS),
                  name='input_pixels')
    ]
    layers = layers + [
              tf.keras.layers.Dense(nodes,
                      kernel_regularizer=regularizer,
                      activation=tf.keras.activations.relu,
                      name='hidden_dense_{}'.format(hno))
              for hno, nodes in enumerate(num_hidden)
    ]
    layers = layers + [
              tf.keras.layers.Dense(len(CLASS_NAMES),
                      kernel_regularizer=regularizer,
                      activation='softmax',
                      name='flower_prob')
    ]

    model = tf.keras.Sequential(layers, name='flower_classification')
```

Notice that we are providing readable names for the layers. This shows up when we print a summary of the model and is also useful to get a layer by name. For example, here is the model where `num_hidden` is [64, 16]:

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_pixels (Flatten)       (None, 150528)            0
_________________________________________________________________
hidden_dense_0 (Dense)       (None, 64)                9633856
_________________________________________________________________
hidden_dense_1 (Dense)       (None, 16)                1040
_________________________________________________________________
flower_prob (Dense)          (None, 5)                 85
=================================================================
Total params: 9,634,981
Trainable params: 9,634,981
Non-trainable params: 0
```

The model, once created, is trained just as before. Unfortunately, as shown in Figure 2-21, the resulting validation accuracy is worse than what was obtained with either the linear model or the neural network.

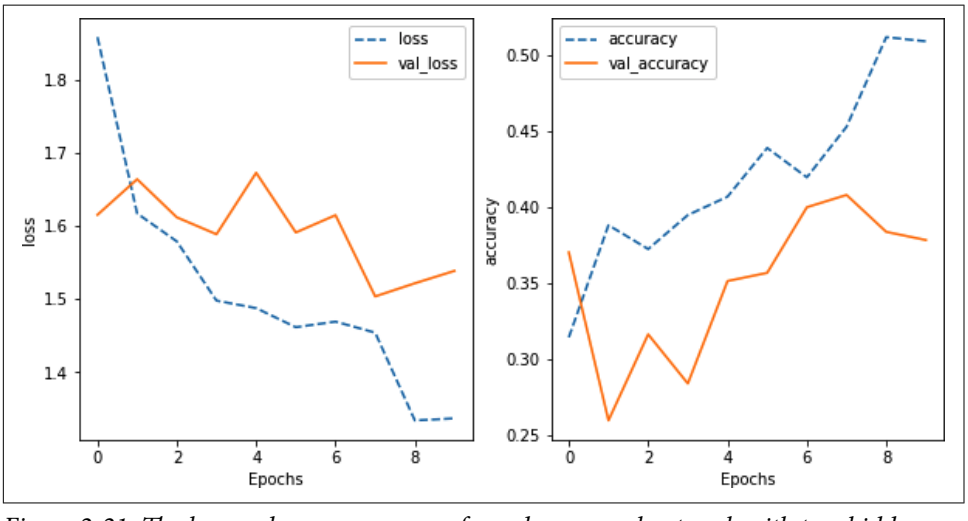


Figure 2-21. The loss and accuracy curves for a deep neural network with two hidden layers.

The 5-flowers dataset is too small for us to take advantage of the additional modeling capability provided by the DNN's extra layer. Recall that we had a similar situation when we started with the neural network. Initially, we did not do better than the linear model, but by adding regularization and lowering the learning rate we were able to get better performance.

Are there some tricks that we can apply to improve the performance of the DNN? Glad you asked! There are two ideas—*dropout layers* and *batch normalization*—that are worth trying.

Dropout

Dropout is one of the oldest regularization techniques in deep learning. At each training iteration, the dropout layer drops random neurons from the network, with a probability p (typically 25% to 50%). In practice, the dropped neurons' outputs are set to zero. The net result is that these neurons will not participate in the loss computation this time around, and they will not get weight updates (see Figure 2-22). Different neurons will be dropped at each training iteration.

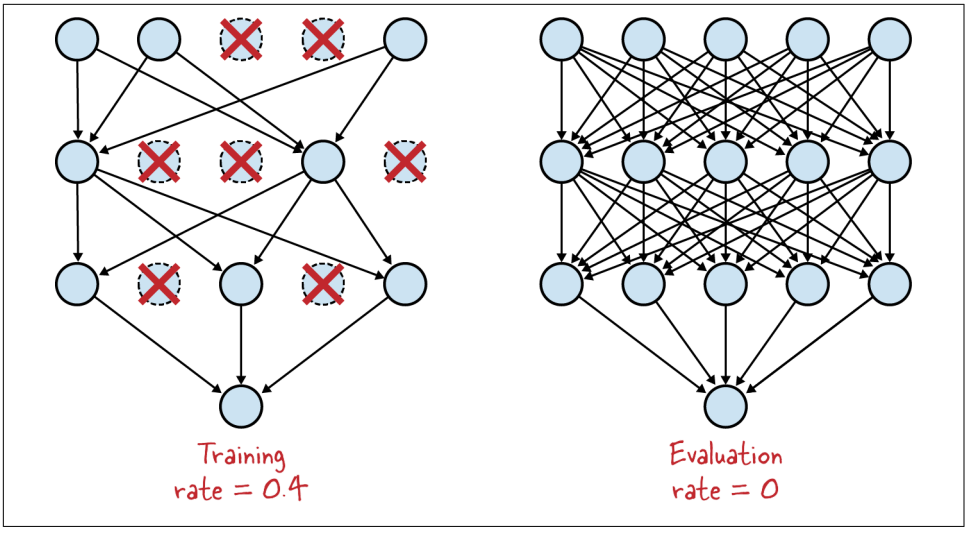


Figure 2-22. Dropout layers are applied during training—here, with a dropout rate of 0.4, 40% of the nodes in the layer are randomly dropped at each step of training.

When testing the performance of the network, all the neurons need to be considered (dropout rate=0). Keras does this automatically, so all you have to do is add a `tf.keras.layers.Dropout` layer. It will automatically have the correct behavior at training and evaluation time: during training, layers are randomly dropped; but during evaluation and prediction, no layers are dropped.

The theory behind dropout is that neural networks have so much freedom between their numerous layers that it is entirely possible for one layer to evolve a bad behavior and for the next layer to compensate for it. This is not an ideal use of neurons. With dropout, there is a high probability that the neurons "fixing" the problem will not be there in a given training round. The bad behavior of the offending layer therefore becomes obvious, and weights evolve toward a better behavior. Dropout also helps spread the information flow throughout the network, giving all weights fairly equal amounts of training, which can help keep the model balanced.

Batch normalization

Our input pixel values are in the range [0, 1], and this is compatible with the dynamic range of the typical activation functions and optimizers. However, once we add a hidden layer, the resulting output values will no longer lie in the dynamic range of the activation function for subsequent layers (see Figure 2-23). When this happens, the neuron's output is zero, and because moving a small amount in either direction makes

no difference, the gradient is zero. There is no way for the network to escape from the dead zone.

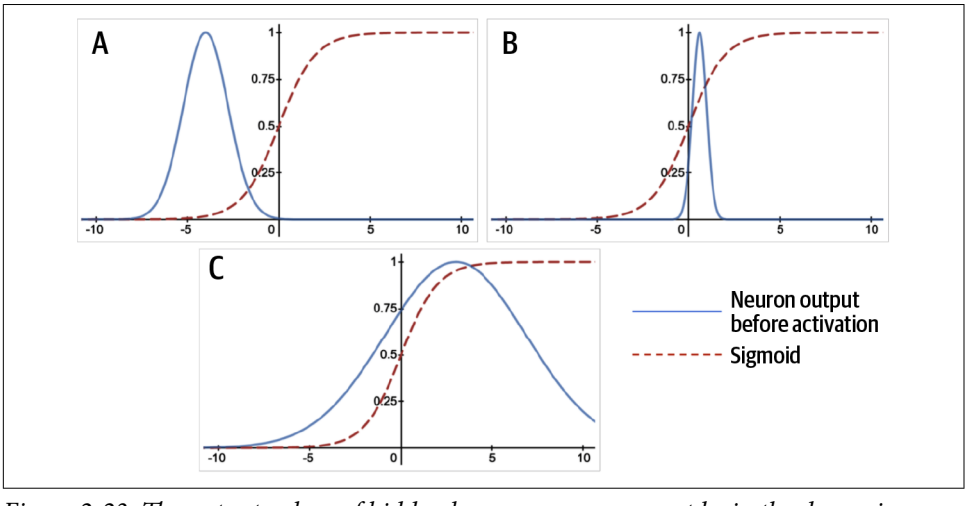


Figure 2-23. The output values of hidden layer neurons may not be in the dynamic range of the activation function. They might be (A) too far to the left (after sigmoid activation, this neuron almost always outputs zero), (B) too narrow (after sigmoid activation, this neuron never outputs a clear 0 or 1), or (C) not too bad (after sigmoid activation, this neuron will output a fair range of outputs between 0 and 1 across a mini-batch).

To fix this, batch normalization normalizes neuron outputs across a training batch of data by subtracting the average and dividing by the standard deviation. However, doing just that could be swinging the pendulum too far in one direction—with a perfectly centered and normally wide distribution everywhere, all neurons would have the same behavior. The trick is to introduce two additional learnable parameters per neuron, called *scale* and *center*, and to normalize the input data to the neuron using these values:

$$normalized = \frac{(input - center)}{scale}$$

This way, the network decides, through machine learning, how much centering and rescaling to apply at each neuron. In Keras, you can selectively use one or the other. For example:

```
tf.keras.layers.BatchNormalization(scale=False, center=True)
```

The problem with batch normalization is that at prediction time you do not have training batches over which you can compute the statistics of your neurons' outputs, but you still need those values. Therefore, during training, neurons' output statistics

are computed across a "sufficient" number of batches using a running exponential average. These stats are then used at inference time.

The good news is that in Keras you can use a `tf.keras.layers.BatchNormalization` layer and all this accounting will happen automatically. When using batch normalization, remember that:

- Batch normalization is performed on the output of a layer before the activation function is applied. So, rather than set `activation='relu'` in the `Dense` layer's constructor, we'd omit the activation function there and then add a separate `Activation` layer.
- If you use `center=True` in batch norm, you do not need biases in your layer. The batch norm offset plays the role of a bias.
- If you use an activation function that is scale-invariant (i.e., does not change shape if you zoom in on it), then you can set `scale=False`. ReLu is scale-invariant. Sigmoid is not.

With dropout and batch normalization, the hidden layers now become:

```
for hno, nodes in enumerate(num_hidden):
    layers.extend([
        tf.keras.layers.Dense(nodes,
                              kernel_regularizer=regularizer,
                              name='hidden_dense_{}'.format(hno)),
        tf.keras.layers.BatchNormalization(scale=False, # ReLU
                              center=False, # have bias in Dense
                              name='batchnorm_dense_{}'.format(hno)),
        # move activation to come after batch norm
        tf.keras.layers.Activation('relu',
                                   name='relu_dense_{}'.format(hno)),
        tf.keras.layers.Dropout(rate=0.4,
                                name='dropout_dense_{}'.format(hno)),

    ])

    layers.append(
        tf.keras.layers.Dense(len(CLASS_NAMES),
                              kernel_regularizer=regularizer,
                              activation='softmax',
                              name='flower_prob')
    )
```

Note that we have moved the activation out of the `Dense` layer and into a separate layer that comes after batch normalization:

```
_________________________________________________________________
hidden_dense_0 (Dense)       (None, 64)                9633856
_________________________________________________________________
batchnorm_dense_0 (BatchNorm (None, 64)                128
_________________________________________________________________
relu_dense_0 (Activation)    (None, 64)                0
_________________________________________________________________
dropout_dense_0 (Dropout)    (None, 64)                0
_________________________________________________________________
```

The resulting training indicates that these two tricks have improved the ability of the model to generalize and to converge faster, as shown in Figure 2-24. We now get an accuracy of 0.48, as opposed to 0.40 without batch norm and dropout. Fundamentally, though, the DNN is not much better than a linear model (0.48 vs. 0.46), because a dense network is not the correct way to go deeper.

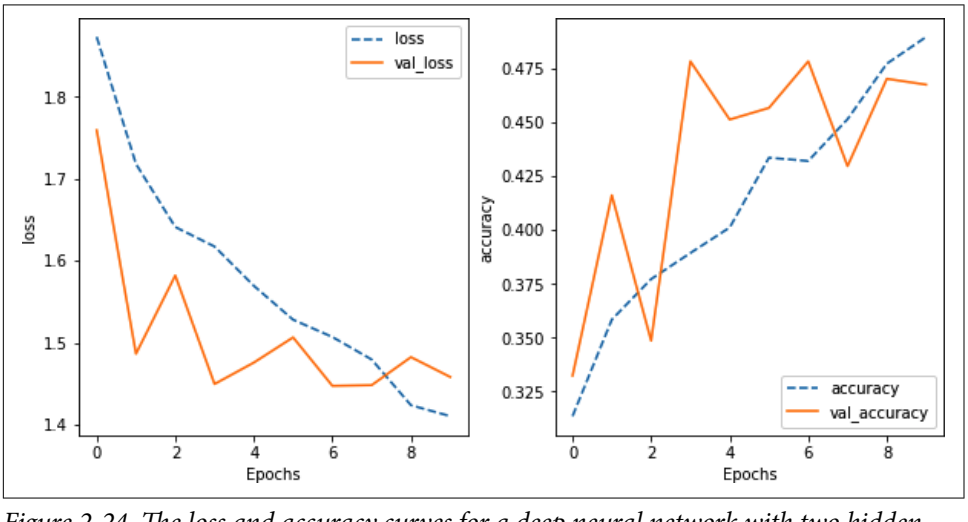


Figure 2-24. The loss and accuracy curves for a deep neural network with two hidden layers with dropout and batch normalization.

The curves are not yet well behaved (note the choppiness of the validation curves). To smooth them out, we will have to experiment with different values of regularization and then do hyperparameter tuning as before. In general, you'll have to experiment with all of these ideas (regularization, early stopping, dropout, batch normalization) for any model you pick. In the rest of the book, we'll simply show the code, but, in practice, model creation will always be followed by a period of experimentation and hyperparameter tuning.

Summary

In this chapter, we explored how to build a simple data pipeline that reads image files and creates 2D floating-point arrays. These arrays were used as inputs into fully connected machine learning models. We started with a linear model, and then added more `Dense` layers. We discovered that regularization was important to limit overfitting, and that changing the learning rate had an impact on learnability.

The models that we built in this chapter did not take advantage of the special structure of images, where adjacent pixels are highly correlated. That is what we will do in Chapter 3. Nevertheless, the tools that we introduced in this chapter for reading images, visualizing them, creating ML models, and predicting using ML models will remain applicable even as the models themselves become more complex. The techniques that you learned about here—hidden layers, changing the learning rate, regularization, early stopping, hyperparameter tuning, dropout, and batch normalization—are used in all the models we discuss in this book.

This chapter introduced a lot of important terminology. For quick reference, a short glossary of terms follows.

Glossary

Accuracy
: An error metric that measures the fraction of correct predictions in a classification model: (TP + TN) / (TP + FP + TN + FN) where, for example, TP is true positives.

Activation function
: A function applied to the weighted sum of the inputs to a node in a neural network. This is the way that nonlinearity is added to a neural network. Common activation functions include ReLU and sigmoid.

AUC
: Area under the curve of true positive rate plotted against false positive rate. The AUC is a threshold-independent error metric.

Batch or mini-batch
: Training is always performed on batches of training data and labels. Doing so helps the algorithm converge. The batch dimension is typically the first dimension of data tensors. For example, a tensor of shape [100, 192, 192, 3] contains 100 images of 192x192 pixels with three values per pixel (RGB).

Batch normalization
: Adding two additional learnable parameters per neuron to normalize the input data to the neuron during training.

Cross-entropy loss
: A special loss function often used in classifiers.

Dense layer
: A layer of neurons where each neuron is connected to all the neurons in the previous layer.

Dropout
: A regularization technique in deep learning where, during each training iteration, randomly chosen neurons from the network are dropped.

Early stopping
: Stopping a training run when the validation set error starts to get worse.

Epoch
: A full pass through the training dataset during training.

Error metric
: The error function comparing neural network outputs to the correct answers. The error on the evaluation dataset is what is reported. Common error metrics include precision, recall, accuracy, and AUC.

Features
: A term used to refer to the inputs of a neural network. In modern image models, the pixel values form the features.

Feature engineering
: The art of figuring out which parts of a dataset (or combinations of parts) to feed into a neural network to get good predictions. In modern image models, no feature engineering is required.

Flattening
: Converting a multidimensional tensor to a 1D tensor that contains all the values.

Hyperparameter tuning
: An "outer" optimization loop where multiple models with different values of model hyperparameters (like learning rate and number of nodes) are trained, and the best of these models chosen. In the "inner" optimization loop, which we call the *training loop*, the model's parameters (weights and biases) are optimized.

Labels
: Another name for "classes," or correct answers in a supervised classification problem.

Learning rate
: The fraction of the gradient by which weights and biases are updated at each iteration of the training loop.

Logits
: The outputs of a layer of neurons before the activation function is applied. The term comes from the *logistic function*, a.k.a. the *sigmoid function*, which used to be the most popular activation function. "Neuron outputs before logistic function" was shortened to "logits."

Loss
: The error function comparing neural network outputs to the correct answers.

Neuron
: The most basic unit of a neural network, which computes the weighted sum of its inputs, adds a bias, and feeds the result through an activation function. The loss on the training dataset is what is minimized during training.

One-hot encoding
: A representation of categorical values as binary vectors. For example, class 3 out of 5 is encoded as a vector of five elements, which are all 0s except the third one, which is a 1: [0 0 1 0 0].

Precision
: An error metric that measures the fraction of true positives in the set of identified positives: TP / (TP + FP).

Recall
: An error metric that measures the fraction of true positives identified among all the positives in the dataset: TP / (TP + FN).

Regularization
: A penalty imposed on weights or model function during training to limit the amount of overfitting. *L1 regularization* drives many of the weight values to zero but is more tolerant of individual large weight values than *L2 regularization*, which tends to drive all the weights to small but nonzero values.

ReLU
: Rectified linear unit. A popular activation function for neurons.

Sigmoid
: An activation function that acts on an unbounded scalar and converts it into a value that lies between [0,1]. It is used as the last step of a binary classifier.

Softmax
: A special activation function that acts on a vector. It increases the difference between the largest component and all others, and also normalizes the vector to have a sum of 1 so that it can be interpreted as a vector of probabilities. Used as the last step in multiclass classifiers.

Tensor
: A tensor is like a matrix but with an arbitrary number of dimensions. A 1D tensor is a vector, a 2D tensor is a matrix, and you can have tensors with three, four, five or more dimensions. In this book, we will use the term tensor to refer to the numerical type that supports GPU-accelerated TensorFlow operations.

Training
: Optimizing the parameters of a machine learning model to attain lower loss on a training dataset.

CHAPTER 3
Image Vision

In Chapter 2, we looked at machine learning models that treat pixels as being independent inputs. Traditional fully connected neural network layers perform poorly on images because they do not take advantage of the fact that adjacent pixels are highly correlated (see Figure 3-1). Moreover, fully connecting multiple layers does not make any special provisions for the 2D hierarchical nature of images. Pixels close to each other work together to create shapes (such as lines and arcs), and these shapes themselves work together to create recognizable parts of an object (such as the stem and petals of a flower).

In this chapter, we will remedy this by looking at techniques and model architectures that take advantage of the special properties of images.

The code for this chapter is in the *03_image_models* folder of the book's GitHub repository. We will provide file names for code samples and notebooks where applicable.

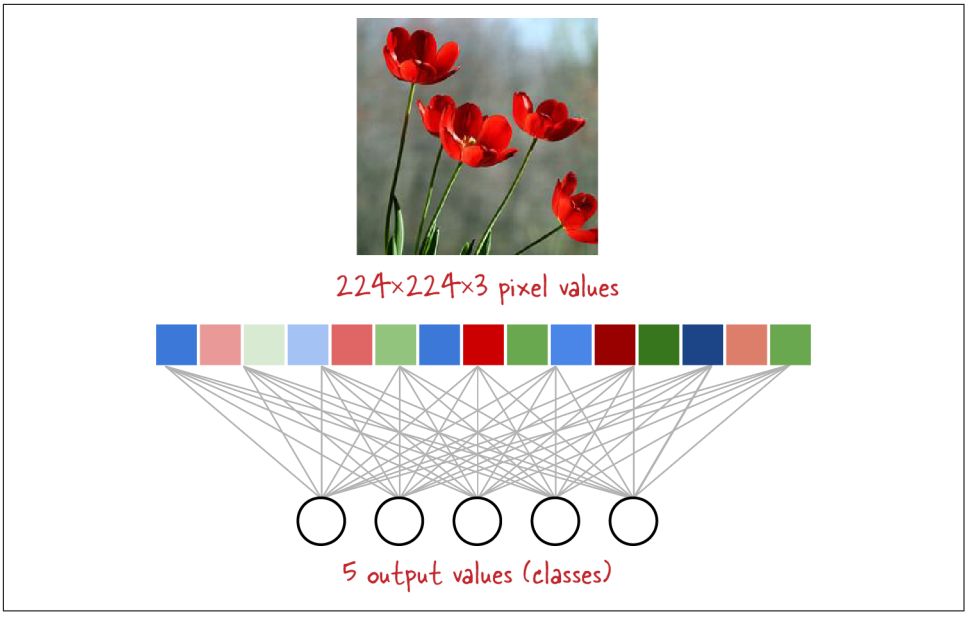


Figure 3-1. Applying a fully connected layer to all the pixels of an image treats the pixels as independent inputs and ignores that images have adjacent pixels working together to create shapes.

Pretrained Embeddings

The deep neural network that we developed in Chapter 2 had two hidden layers, one with 64 nodes and the other with 16 nodes. One way to think about this network architecture is shown in Figure 3-2. In some sense, all the information contained in the input image is being represented by the penultimate layer, whose output consists of 16 numbers. These 16 numbers that provide a representation of the image are called an *embedding*. Of course, earlier layers also capture information from the input image, but those are typically not used as embeddings because they are missing some of the hierarchical information.

In this section, we will discuss how to create an embedding (as distinct from a classification model), and how to use the embedding to train models on different datasets using two different approaches, transfer learning and fine-tuning.

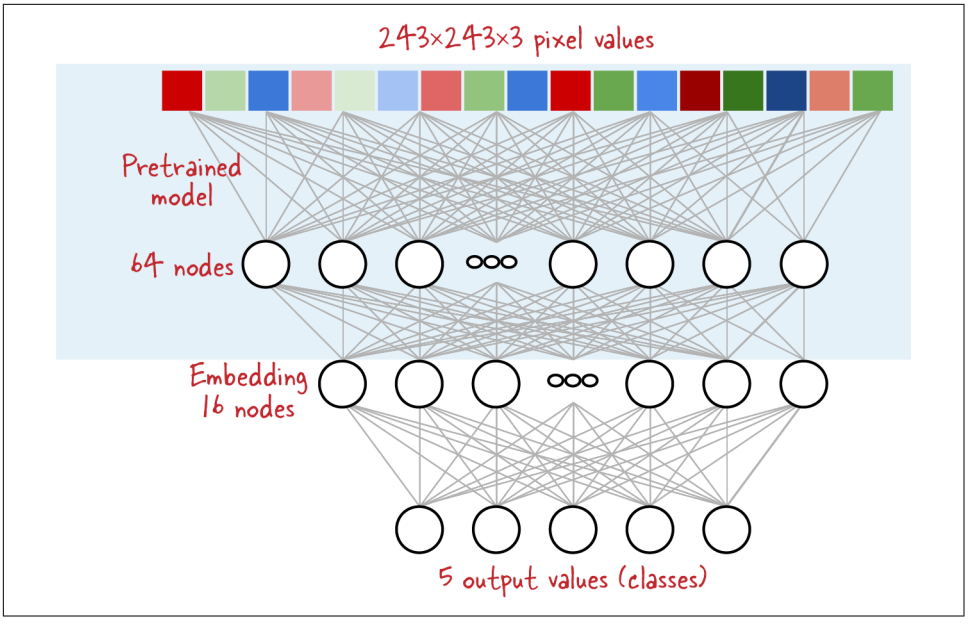


Figure 3-2. The 16 numbers that form the embedding provide a representation of all the information in the entire image.

Pretrained Model

The embedding is created by applying a set of mathematical operations to the input image. Recall that we reiterated in Chapter 2 that the model accuracy that we were getting, on the order of 0.45, was low because our dataset wasn't large enough to support the many millions of trainable weights in our fully connected deep learning model. What if we were to repurpose the embedding creation part from a model that has been trained on a much larger dataset? We can't repurpose the whole model, because that model will not have been trained to classify flowers. However, we can throw away the last layer, or *prediction head*, of that model and replace it with our own. The repurposed part of the model can be *pretrained* from a very large, general-purpose dataset and the knowledge can then be *transferred* to the actual dataset that we want to classify. Looking back to Figure 3-2, we can replace the 64-node layer in the box marked "pretrained model" with the first set of layers of a model that has been trained on a much larger dataset.

Pretrained models are models that are trained on large datasets and made available to be used as a way to create embeddings. For example, the MobileNet model is a model with 1–4 million parameters that was trained on the ImageNet (ILSVRC) dataset, which consists of millions of images corresponding to hundreds of categories that were scraped from the web. The resulting embedding therefore has the ability to efficiently compress the information found in a wide variety of images. As long as the

images we want to classify are similar in nature to the ones that MobileNet was trained on, the embeddings from MobileNet should give us a great pretrained embedding that we can use as a starting point to train a model on our own smaller dataset.

A pretrained MobileNet is available on TensorFlow Hub, and we can easily load it as a Keras layer by passing in the URL to the trained model:

```
import tensorflow_hub as hub
huburl= "https://tfhub.dev/google/imagenet/\
    mobilenet_v2_100_224/feature_vector/4"
hub.KerasLayer(
    handle=huburl,
    input_shape=(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS),
    trainable=False,
    name='mobilenet_embedding')
```

In this code snippet, we imported the package `tensorflow_hub` and created a `hub.KerasLayer`, passing in the URL and the input shape of our images. Critically, we specify that this layer is not trainable, and should be assumed to be pretrained. By doing so, we ensure that its weights will not be modified based on the flowers data; it will be read-only.

Transfer Learning

The rest of the model is similar to the DNN models that we created previously. Here's a model that uses the pretrained model loaded from TensorFlow Hub as its first layer (the full code is available in *03a_transfer_learning.ipynb*):

```
layers = [
    hub.KerasLayer(..., name='mobilenet_embedding'),
    tf.keras.layers.Dense(units=16,
                          activation='relu',
                          name='dense_hidden'),
    tf.keras.layers.Dense(units=len(CLASS_NAMES),
                          activation='softmax',
                          name='flower_prob')
]
model = tf.keras.Sequential(layers, name='flower_classification')
...
```

The resulting model summary is as follows:

```
Model: "flower_classification"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
mobilenet_embedding (KerasLa (None, 1280)              2257984
_________________________________________________________________
dense_hidden (Dense)         (None, 16)                20496
_________________________________________________________________
flower_prob (Dense)          (None, 5)                 85
```

```
=================================================================
Total params: 2,278,565
Trainable params: 20,581
Non-trainable params: 2,257,984
```

Note that the first layer, which we called `mobilenet_embedding`, has 2.26 million parameters, but they are not trainable. Only 20,581 parameters are trainable: 1,280 * 16 weights + 16 biases = 20,496 from the hidden dense layer, plus 16 * 5 weights + 5 biases = 85 from the dense layer to the five output nodes. So despite the 5-flowers dataset not being large enough to train millions of parameters, it is large enough to train just 20K parameters.

This process of training a model by replacing its input layer with an image embedding is called *transfer learning*, because we have transferred the knowledge learned from a much larger dataset by the MobileNet creators to our problem.

Because we are replacing the input layer of our model with the Hub layer, it's important to make sure that our data pipeline provides data in the format expected by the Hub layer. All image models in TensorFlow Hub use a common image format and expect pixel values as floats in the range [0,1). The image-reading code that we used in Chapter 2 scales the JPEG images to lie within this range, so we're OK.

Pretrained Models in Keras

We've just showed you how to load a pretrained model from TensorFlow Hub. The most popular pretrained models are available directly in Keras. They can be loaded by instantiating the corresponding class in `tf.keras.applications.*`. For example:

```
pretrained_model = tf.keras.applications.MobileNetV2(
  weights='imagenet',include_top=False,
  input_shape=[IMG_HEIGHT, IMG_WIDTH, 3])
pretrained_model.trainable = False  # For transfer learning
```

The following code uses the pretrained model to build a custom classifier, by attaching a custom classification head to it:

```
model = tf.keras.Sequential([
  # convert image format from int [0,255]
  # to the format expected by this model
  tf.keras.layers.Lambda(
    lambda data: tf.keras.applications.mobilenet.preprocess_input(
      tf.cast(data, tf.float32)),
    input_shape=[IMG_HEIGHT, IMG_WIDTH, 3]),
  pretrained_model,
  tf.keras.layers.GlobalAveragePooling2D(),
  tf.keras.layers.Dense(256, activation='relu'),
  tf.keras.layers.Dense(len(CLASSES),
  activation='softmax')
])
```

Notice how the code snippet handles the pretrained model's expected inputs and outputs:

1. Every model in `tf.keras.applications.*` expects its input images to have pixel values in a specific range, such as [0, 1] or [–1, 1]. A format conversion function named `tf.keras.applications.<MODEL_NAME>.preprocess_input()` is provided for every model. It converts images with pixel values that are floats in the range [0, 255] into the pixel format expected by the pretrained model. If you load images using the `tf.io.decode_image()` operation, which returns pixels with a `uint8` format in the range [0, 255], a cast to float is necessary before applying `preprocess_input()`.

This is different from the image format convention used in TensorFlow Hub. All the image models in TensorFlow Hub expect pixel values as floats in the range [0, 1). The easiest way to obtain images in that format is to use `tf.io.decode_image()` followed by `tf.image.convert_image_dtype(..., tf.float32)`.

2. With the option `include_top=False`, all the models in `tf.keras.applications.*` return a 3D feature map. It's the user's responsibility to compute a 1D feature vector from it so that a classification head with dense layers can be appended. You can use `tf.keras.layers.GlobalAveragePooling2D()` or `tf.keras.layers.Flatten()` for this purpose.

This is again different from the way models in TensorFlow Hub typically return embeddings. All models in TensorFlow Hub that include `feature_vector` in their name already return a 1D feature vector, not a 3D feature map. A dense layer can be added immediately after them to implement a custom classification head.

Training this model is identical to training the DNN in the previous section (see *03a_transfer_learning.ipynb* in the GitHub repository for details). The resulting loss and accuracy curves are shown in Figure 3-3.

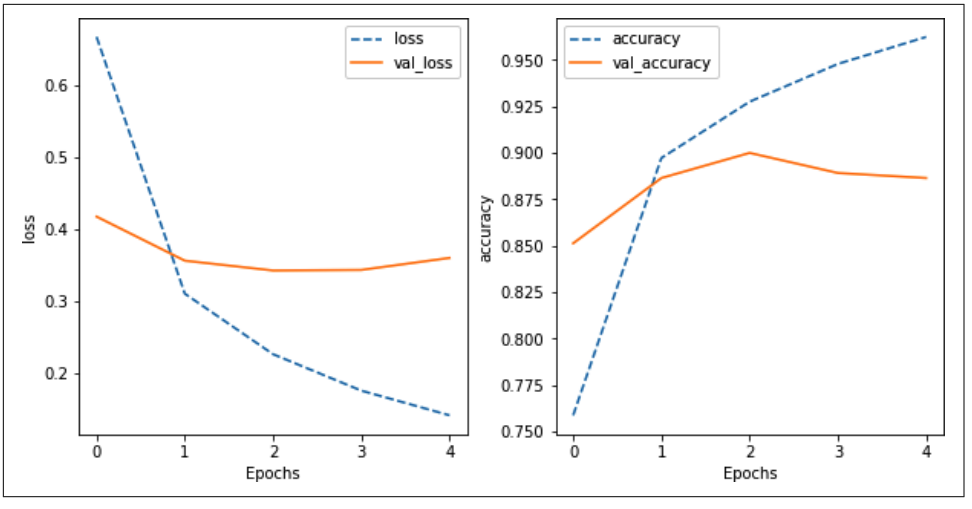


Figure 3-3. The loss and accuracy curves for a deep neural network with two hidden layers with dropout and batch normalization.

Rather impressively, we get an accuracy of 0.9 using transfer learning (see Figure 3-4), whereas we got to only 0.48 when training a fully connected deep neural network from scratch on our data. Transfer learning is what we recommend any time your dataset is relatively small. Only when your dataset starts to exceed about five thousand images *per label* should you start to consider training from scratch. Later in this chapter, we will see techniques and architectures that allow us to get even higher accuracies provided we have a large dataset and can train from scratch.

The probability associated with the daisy prediction for the first image in the second row might come as a surprise. How can the probability be 0.41? Shouldn't it be greater than 0.5? Recall that this is not a binary prediction problem. There are five possible classes, and if the output probabilities are [0.41, 0.39, 0.1, 0.1, 0.1], the `argmax` will correspond to daisy and the probability will be 0.41.

Figure 3-4. Predictions by the MobileNet transfer learning model on some of the images in the evaluation dataset.

Fine-Tuning

During transfer learning, we took all the layers that comprise MobileNet and used them as is. We did so by making the layers non-trainable. Only the last two dense layers were tuned on the 5-flowers dataset.

In many instances, we might be able to get better results if we allow our training loop to also adapt the pretrained layers. This technique is called *fine-tuning*. The pretrained weights are used as initial values for the weights of the neural network (normally, neural network training starts with the weights initialized to random values).

In theory, all that is needed to switch from transfer learning to fine-tuning is to flip the `trainable` flag from `False` to `True` when loading a pretrained model and train on your data. In practice, however, you will often notice training curves like the one in Figure 3-5 when fine-tuning a pretrained model.

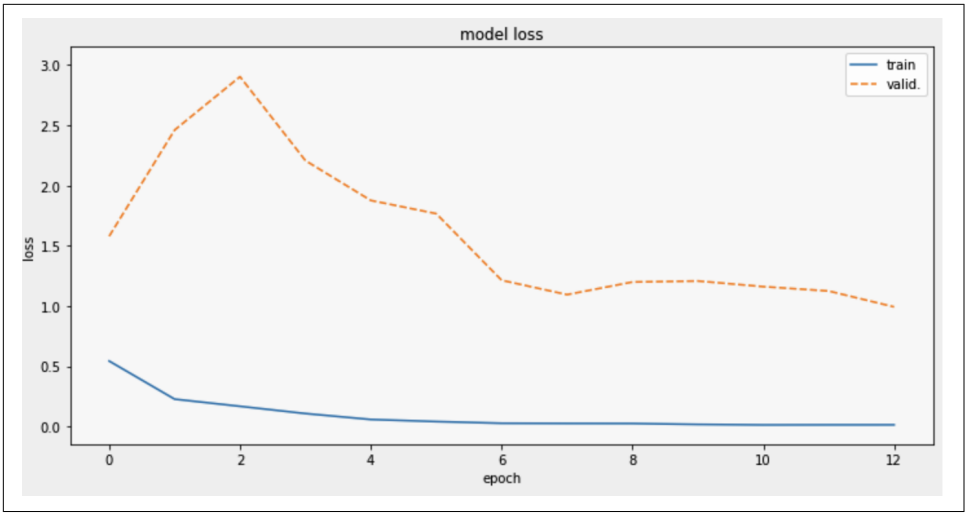


Figure 3-5. The training and validation loss curves when fine-tuning with a badly chosen learning rate schedule.

The training curve here shows that the model mathematically converges. However, its performance on the validation data is poor and initially gets worse before somewhat recovering. With a learning rate set too high, the pretrained weights are being changed in large steps and all the information learned during pretraining is lost. Finding a learning rate that works can be tricky—set the learning rate too low and convergence is very slow, too high and pretrained weights are lost.

There are two techniques that can be used to solve this problem: a learning rate schedule and layer-wise learning rates. The code showcasing both techniques is available in *03b_finetune_MOBILENETV2_flowers5.ipynb*.

Learning rate schedule

The most traditional learning rate schedule when training neural networks is to have the learning rate start high and then decay exponentially throughout the training. When fine-tuning a pretrained model, a warm-up ramp period can be added (see Figure 3-6).

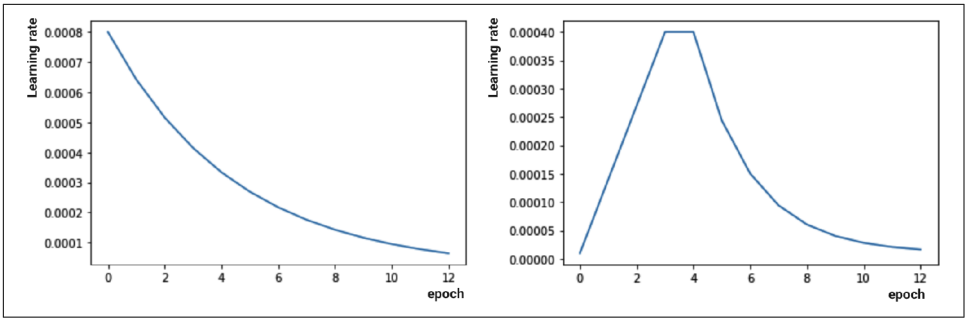


Figure 3-6. On the left, a traditional learning rate schedule with exponential decay; on the right, a learning rate schedule that features a warm-up ramp, which is more appropriate for fine-tuning.

Figure 3-7 shows the loss curves with this new learning rate schedule.

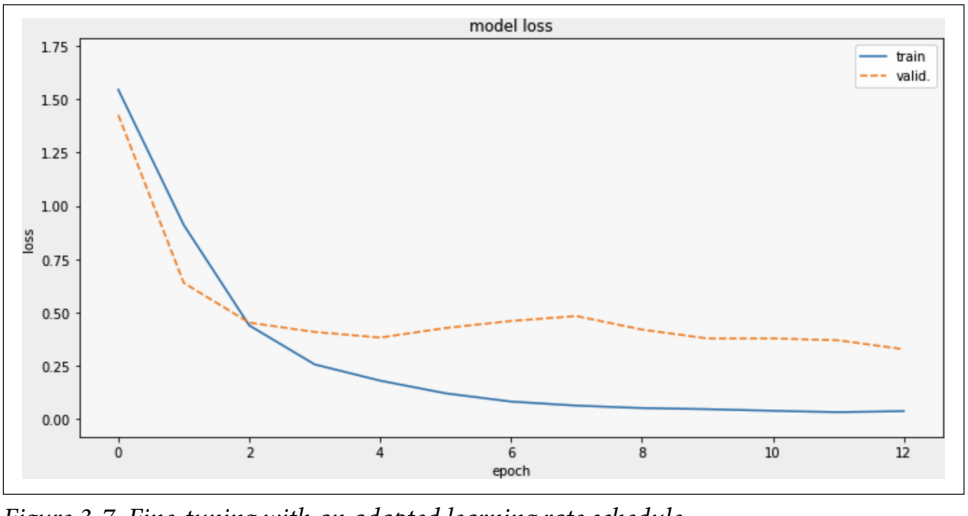


Figure 3-7. Fine-tuning with an adapted learning rate schedule.

Notice that there is still a hiccup on the validation loss curve, but it is nowhere as bad as previously (compare with Figure 3-5). This leads us to the second way to choose learning rates for fine-tuning.

Differential learning rate

Another good trade-off is to apply a *differential learning rate*, whereby we use a low learning rate for the pretrained layers and a normal learning rate for the layers of our custom classification head.

In fact, we can extend the idea of a differential learning rate within the pretrained layers themselves—we can multiply the learning rate by a factor that varies based on

layer depth, gradually increasing the per-layer learning rate and finishing with the full learning rate for the classification head.

In order to apply a complex differential learning rate like this in Keras, we need to write a custom optimizer. But fortunately, an open source Python package called AdamW exists that we can use by specifying a learning rate multiplier for different layers (see *03_image_models/03b_finetune_MOBILENETV2_flowers5.ipynb* in the GitHub repository for the complete code):

```
mult_by_layer={
    'block1_': 0.1,
    'block2_': 0.15,
    'block3_': 0.2,
    ... # blocks 4 to 11 here
    'block12_': 0.8,
    'block13_': 0.9,
    'block14_': 0.95,
    'flower_prob': 1.0, # for the classification head
}

optimizer = AdamW(lr=LR_MAX, model=model,
                  lr_multipliers=mult_by_layer)
```

How did we know what the names of the layers in the loaded pretrained model were? We ran the code without any name at first, with `lr_multipliers={}`. The custom optimizer prints the names of all the layers when run. We then found a substring of the layer names that identified the depth of the layer in the network. The custom optimizer matches layer names by the substrings passed to its `lr_multipliers` argument.

With both the per-layer learning rate and a learning rate schedule with a ramp-up, we can push the accuracy of a fine-tuned MobileNetV2 on the `tf_flowers` (5-flowers) dataset to 0.92, versus 0.91 with the ramp-up only and 0.9 with transfer learning only (see the code in *03b_finetune_MOBILENETV2_flowers5.ipynb*).

The gains from fine-tuning here are small because the `tf_flowers` dataset is tiny. We need a more challenging benchmark for the advanced architectures we are about to explore. In the rest of this chapter, we will use the *104 flowers* dataset.

The 104 Flowers Dataset

The 104 flowers dataset has more than 23,000 labeled images of 104 kinds of flowers. It was assembled for the "Petals to the Metal" competition on Kaggle from various publicly available image datasets. Some samples are shown in Figure 3-8.

Figure 3-8. An excerpt from the 104 flowers dataset.

We will use this dataset for the remainder of this chapter. Being a larger dataset, it also requires more resources to train on. That is why all the examples in the rest of the chapter are set up to run both on GPUs and TPUs (more information on TPUs is provided in Chapter 7). The data is stored in TFRecords, for reasons we explain in Chapter 5.

This dataset is also challenging because it is heavily imbalanced, with thousands of images in some categories and less than a hundred examples in others. This reflects how datasets are found in real life. Accuracy—i.e., the percentage of correctly classified images—is not a good metric for an imbalanced dataset. So, *precision*, *recall*, and *F1* score will be used instead. As you learned in Chapter 2, the precision score for the category "daisy" is the fraction of daisy predictions that are correct, while the recall score is the fraction of daisies in the dataset that are correctly classified. The F1 score is the harmonic mean of the two. The overall precision is the weighted average of the precision of all the categories, weighted by the number of instances in each category. More information about these metrics can be found in Chapter 8.

The GitHub repository contains three notebooks experimenting with these fine-tuning techniques on the larger 104 flowers dataset. The results are presented in Table 3-1. For this task we used Xception, a model with more weights and layers than MobileNet, because the 104 flowers dataset is larger and can support this larger model. As you can see, a learning rate ramp-up or a per-layer differential learning rate is not strictly necessary, but in practice it makes the convergence more stable and makes it easier to find working learning rate parameters.

Table 3-1. Summary of results obtained by a larger model (Xception) fine-tuned on the larger 104 flowers dataset

Notebook name	LR ramp-up	Differential LR	Mean F1 score across five runs	Standard deviation across five runs	Notes
lr_decay_xception	No	No	0.932	0.004	Good, relatively low variance
lr_ramp_xception	Yes	No	0.934	0.007	Very good, high variance
lr_layers_lr_ramp_xception	Yes	Yes	0.936	0.003	Best, nice low variance

So far, we have used MobileNet and Xception for transfer learning and fine-tuning, but these models are black boxes as far as we are concerned. We do not know how many layers they have, or what those layers consist of. In the next section, we will discuss a key concept, *convolution*, that helps these neural networks work well at extracting the semantic information content of an image.

Convolutional Networks

Convolutional layers were designed specifically for images. They operate in two dimensions and can capture shape information; they work by sliding a small window, called a *convolutional filter*, across the image in both directions.

Convolutional Filters

A typical 4x4 filter will have independent filter weights for each of the channels of the image. For color images with red, green, and blue channels, the filter will have 4 * 4 * 3 = 48 learnable weights in total. The filter is applied to a single position in the image by multiplying the pixel values in the neighborhood of that position by filter weights and summing them as shown in Figure 3-9. This operation is called the tensor *dot product*. Computing the dot product at each position in the image by sliding the filter across the image is called a *convolution*.

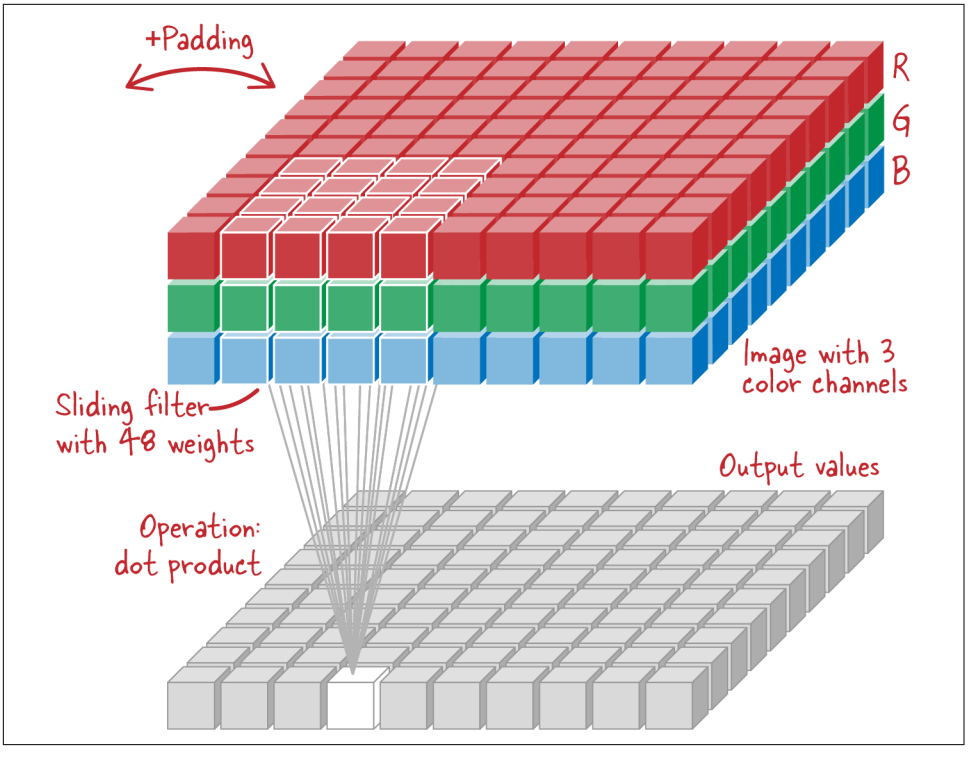


Figure 3-9. Processing an image with a single 4x4 convolutional filter—the filter slides across the image in both directions, producing one output value at each position.

Why Do Convolutional Filters Work?

Convolutional filters have been used in image processing for a long time. They can achieve many different effects. For example, a filter where all the weights are the same is a "smoothing" filter (because each pixel within a window has an equal contribution for the resulting output pixel) and yields the second panel shown in Figure 3-10. Organizing the weights in other specific ways can create edge and intensity detectors (for details, see *03_image_models/diagrams.ipynb* in the GitHub repository).

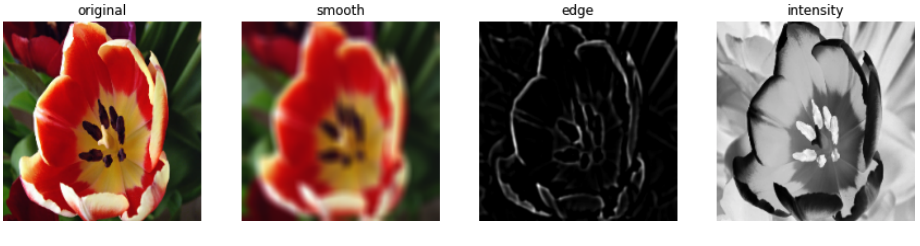


Figure 3-10. The effects of different convolutional filters.

The more interesting filters, like the edge filter in Figure 3-10 (panel 3), use correlations and anti-correlations of adjacent pixels to compute new information about the image. Indeed, adjacent pixels tend to be highly correlated and work together to create what we call *textures* and *edges* at small scales and *shapes* at higher scales. That is where the information of the image is encoded, and that is also what convolutional filters are well equipped to detect.

A convolutional neural network filter, therefore, makes it possible for the machine learning model to learn the arrangement of weights that best picks up pertinent details from the training data. The network will learn whatever combination of weights will minimize the loss.

Another advantage of convolutional filters is that a 5x5x3 filter has only 75 weights, and the same weights are slid across the entire image. Contrast this with a fully connected network layer, where an image that is 200x200x3 will end up with 120K weights *per node* in the next layer! Convolutional filters can therefore help limit the complexity of the neural network. Since the size of the dataset we require for training is related to the number of trainable parameters, using convolutional filters allows us to use our training data more effectively.

A single convolutional filter can process an entire image with very few learnable parameters—so few, in fact, that it will not be able to learn and represent enough of the complexities of the image. Multiple such filters are needed. A convolutional layer typically contains tens or hundreds of similar filters, each with its own independent learnable weights (see Figure 3-11). They are applied to the image in succession, and each produces a *channel* of output values. The output of a convolutional layer is a multichannel set of 2D values. Notice that this output has the same number of dimensions as the input image, which was already a three-channel set of 2D pixel values.

Understanding the structure of a convolutional layer makes it easy to compute its number of learnable weights, as you can see in Figure 3-12. This diagram also introduces the schematic notation of convolutional layers that will be used for the models in this chapter.

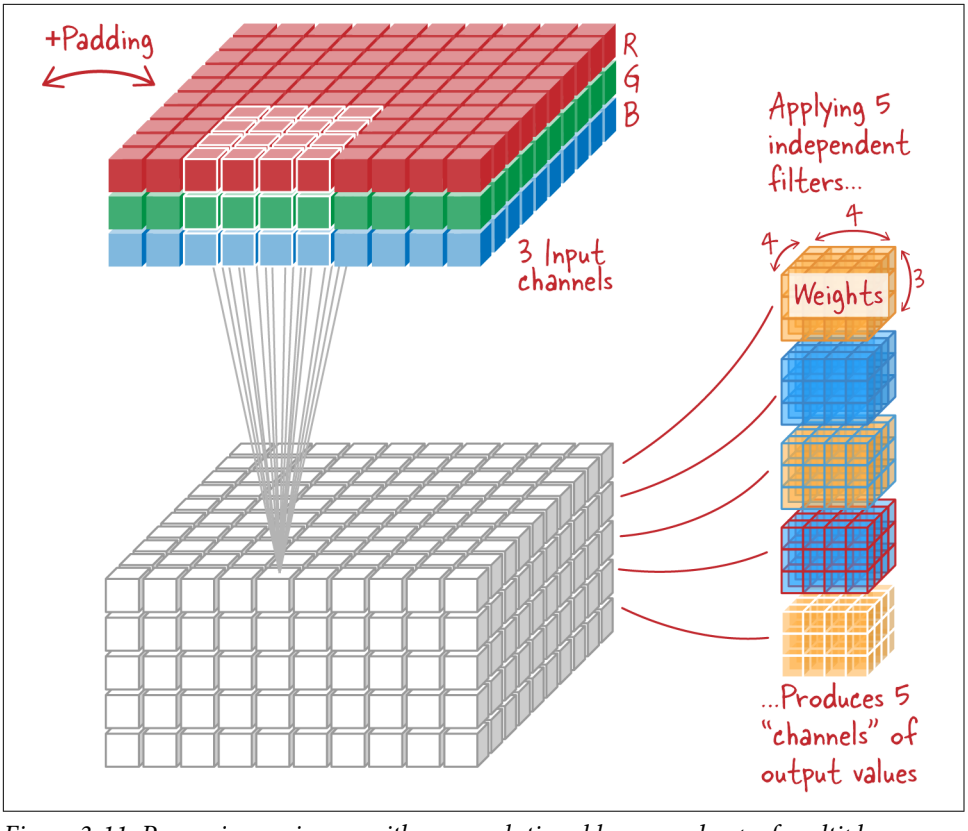


Figure 3-11. Processing an image with a convolutional layer made up of multiple convolutional filters—all filters are of the same size (here, 4x4x3) but have independent learnable weights.

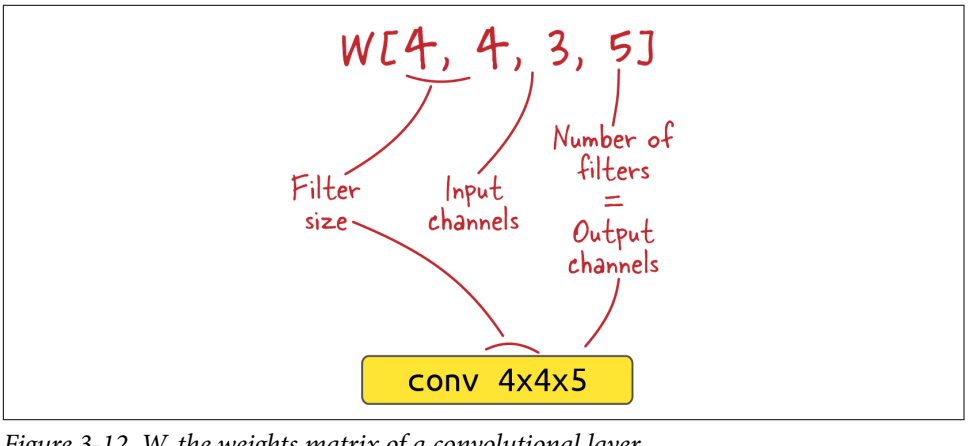


Figure 3-12. W, the weights matrix of a convolutional layer.

In this case, with 5 filters applied, the total number of learnable weights in this convolutional layer is 4 * 4 * 3 * 5 = 240.

Convolutional layers are available in Keras:

```
tf.keras.layers.Conv2D(filters,
                       kernel_size,
                       strides=(1, 1),
                       padding='valid',
                       activation=None)
```

The following is a simplified description of the parameters (see the Keras documentation for full details):

`filters`
: The number of independent filters to apply to the input. This will also be the number of output channels in the output.

`kernel_size`
: The size of each filter. This can be a single number, like 4 for a 4x4 filter, or a pair like (4, 2) for a rectangular 4x2 filter.

`strides`
: The filter slides across the input image in steps. The default step size is 1 pixel in both directions. Using a larger step will skip input pixels and produce fewer output values.

`padding`
: `'valid'` for no padding or `'same'` for zero-padding at the edges. If filters are applied to inputs with `'valid'` padding, convolution is carried out only if all the pixels within the window are valid, so boundary pixels get ignored. Therefore, the output will be slightly smaller in the x and y directions. The value `'same'` enables zero-padding of the input to make sure that outputs have the same width and height as the input.

`activation`
: Like any neural network layer, a convolutional layer can be followed by an activation (nonlinearity).

The convolutional layer illustrated in Figure 3-11, with five 4x4 filters, input padding, and the default stride of 1 in both directions, can be implemented as follows:

```
tf.keras.layers.Conv2D(filters=5, kernel_size=4, padding='same')
```

4D tensors are expected as inputs and outputs of convolutional layers. The first dimension is the batch size, so the full shape is [batch, height, width, channels]. For example, a batch of 16 color (RGB) images of 512x512 pixels would be represented as a tensor with dimensions [16, 512, 512, 3].

Stacking Convolutional Layers

As described in the previous section, a generic convolutional layer takes a 4D tensor of shape [batch, height, width, channels] as an input and produces another 4D tensor as an output. For simplicity, we will ignore the batch dimension in our diagrams and show what happens to a single 3D image of shape [height, width, channels].

A convolutional layer transforms a "cube" of data into another "cube" of data, which can in turn be consumed by another convolutional layer. Convolutional layers can be stacked as shown in Figure 3-13.

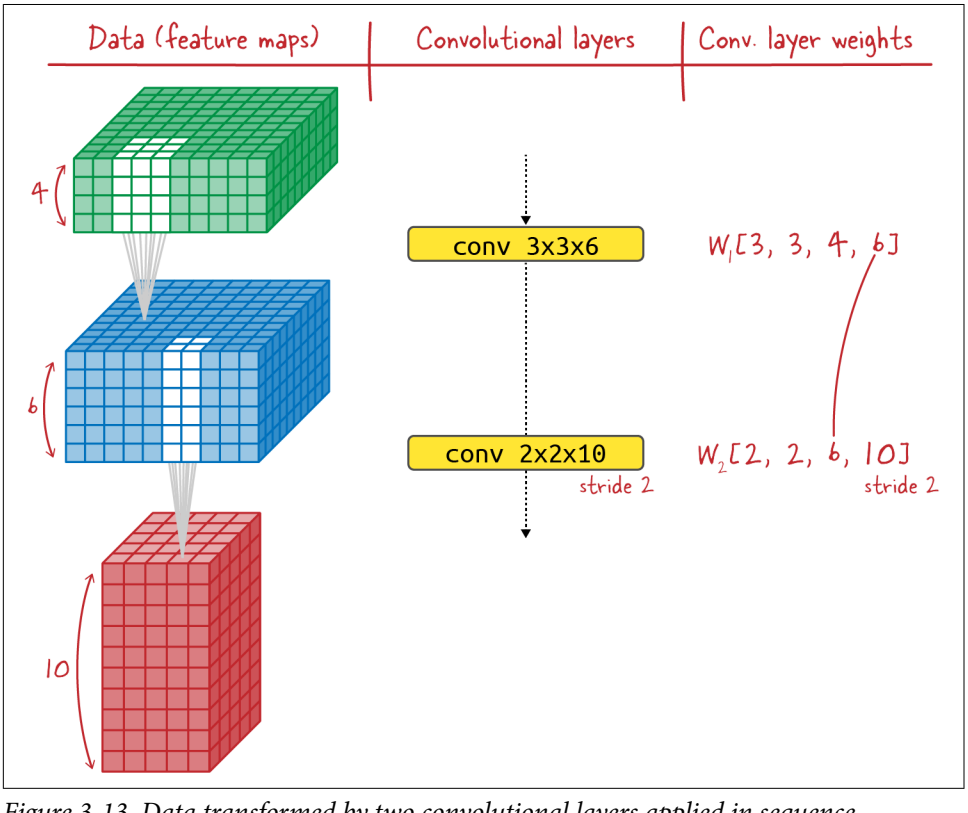


Figure 3-13. Data transformed by two convolutional layers applied in sequence. Learnable weights are shown on the right. The second convolutional layer is applied with a stride of 2 and has six input channels, matching the six output channels of the previous layer.

Figure 3-13 shows how the data is transformed by two convolutional layers. Starting from the top, the first layer is a 3x3 filter applied to an input with four channels of data. The filter is applied to the input six times, each time with different filter weights, resulting in six channels of output values. This in turn is fed into a second

convolutional layer using 2x2 filters. Notice that the second convolutional layer uses a stride of 2 (every other pixel) when applying its filters to obtain fewer output values (in the horizontal plane).

Pooling Layers

The number of filters applied in each convolutional layer determines the number of channels in the output. But how can we control the amount of data in each channel? The goal of a neural network is usually to distill information from the input image, consisting of millions of pixels, to a handful of classes. So, we will need layers that can combine or downsample the information in each channel.

The most commonly used downsampling operation is 2x2 *max pooling*. With max pooling, only the maximum value is retained for each group of four input values from a channel (Figure 3-14). *Average pooling* works in a similar way, averaging the four values instead of keeping the max.

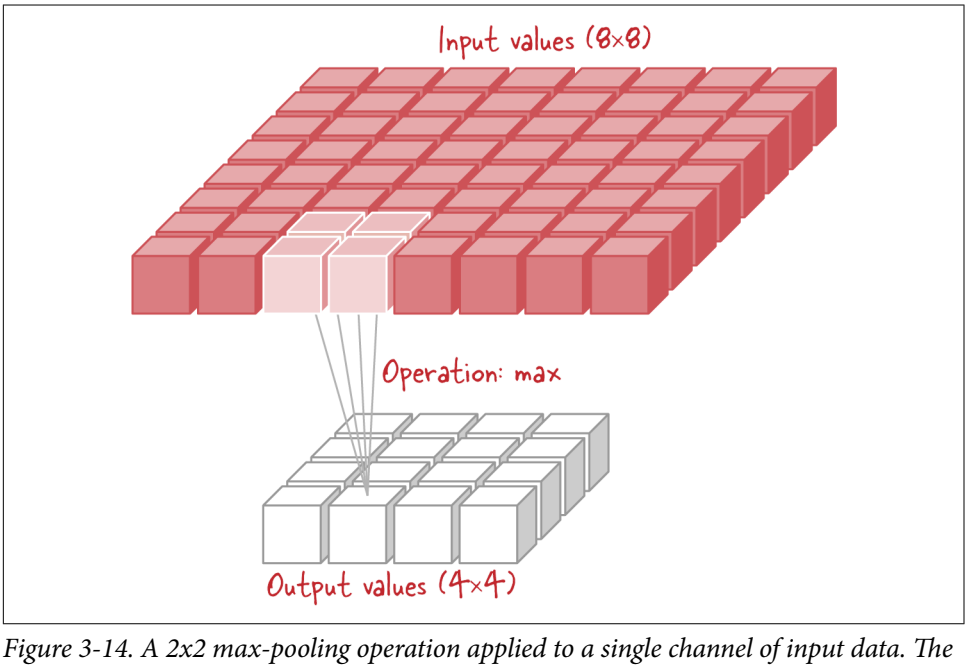


Figure 3-14. A 2x2 max-pooling operation applied to a single channel of input data. The max is taken for every group of 2x2 input values and the operation is repeated every two values in each direction (stride 2).

Note that max-pooling and average-pooling layers do not have any trainable weights. They are purely size adjustment layers.

There is an interesting physical explanation of why max-pooling layers work well with convolutional layers in neural networks. Convolutional layers are series of

trainable filters. After training, each filter specializes in matching some specific image feature. The first layer in a convolutional neural network reacts to pixel combinations in the input image, but subsequent layers react to combinations of features from the previous layers. For example, in a neural network trained to recognize cats, the first layer reacts to basic image components like horizontal and vertical lines or the texture of fur. Subsequent layers react to specific combinations of lines and fur to recognize pointy ears, whiskers, or cat eyes. Even later layers detect a combination of pointy ears + whiskers + cat eyes as a cat head. A max-pooling layer only keeps values where some feature *X* was detected with maximum intensity. If the goal is to reduce the number of values but keep the ones most representative of what was detected, it makes sense.

Pooling layers and convolutional layers also have different effects on the locations of detected features. A convolutional layer returns a feature map with its high values located where its filters detected something significant. Pooling layers, on the other hand, reduce the resolution of the feature maps and make the location information less accurate. Sometimes location or relative location is important, such as eyes usually being located above the nose in a face. Convolutions do produce location information for other layers further along in the network to work with. At other times, however, locating a feature is not the goal—for instance, in a flower classifier, where you want to train the model to recognize flowers in an image wherever they are. In such a case, when training for location invariance, pooling layers help blur the location information to some extent, but not completely. The network will have to be trained on images showing flowers in many different locations if it is to become truly location-agnostic. Data augmentation methods like random crops of the image can be used to force the network to learn this location invariance. Data augmentation is covered in Chapter 6.

A second option for downsampling channel information is to apply convolutions with a stride of 2 or 3 instead of 1. The convolutional filters then slide over the input image by steps of 2 or 3 pixels in each direction. This mechanically produces a certain size of output values, as shown in Figure 3-15.

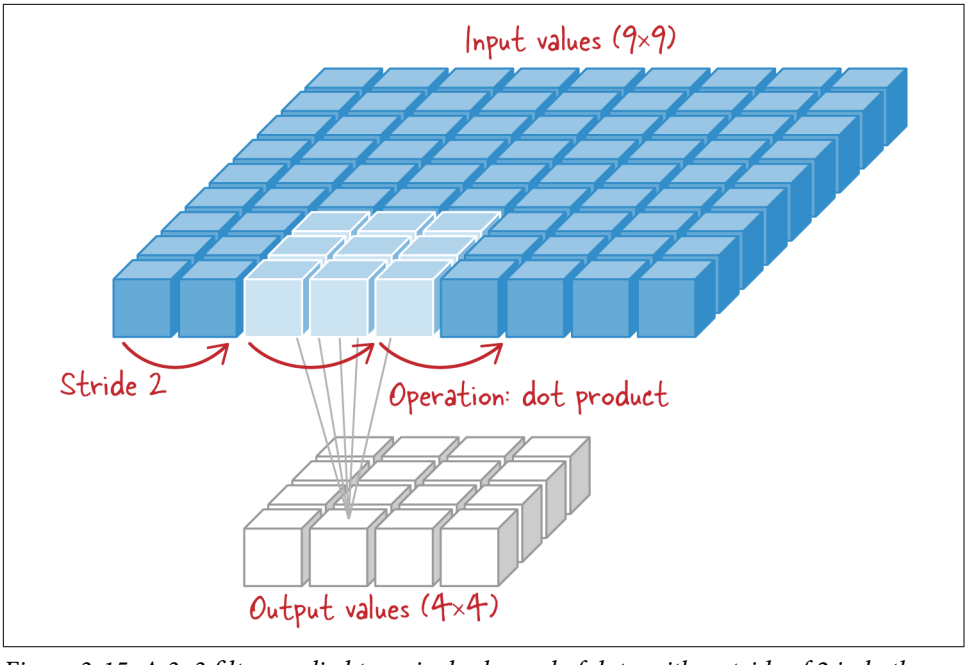


Figure 3-15. A 3x3 filter applied to a single channel of data with a stride of 2 in both directions and without padding. The filter jumps by 2 pixels at a time.

We are now ready to assemble these layers into our first convolutional neural classifier.

AlexNet

The simplest convolutional neural network architecture is a mix of convolutional layers and max-pooling layers. It transforms each input image into a final rectangular prism of values, usually called a *feature map*, which is then fed into a number of fully connected layers and, finally, a softmax layer to compute class probabilities.

AlexNet, introduced in a 2012 paper by Alex Krizhevsky et al. and shown in Figure 3-16, is such an architecture. It was designed for the ImageNet competition, which asked participants to classify images into one thousand categories (car, flower, dog, etc.) based on a training dataset of more than a million images. AlexNet was one of the earliest successes in neural image classification, exhibiting a dramatic improvement in accuracy and proving that deep learning was much better able to address computer vision problems than existing techniques.

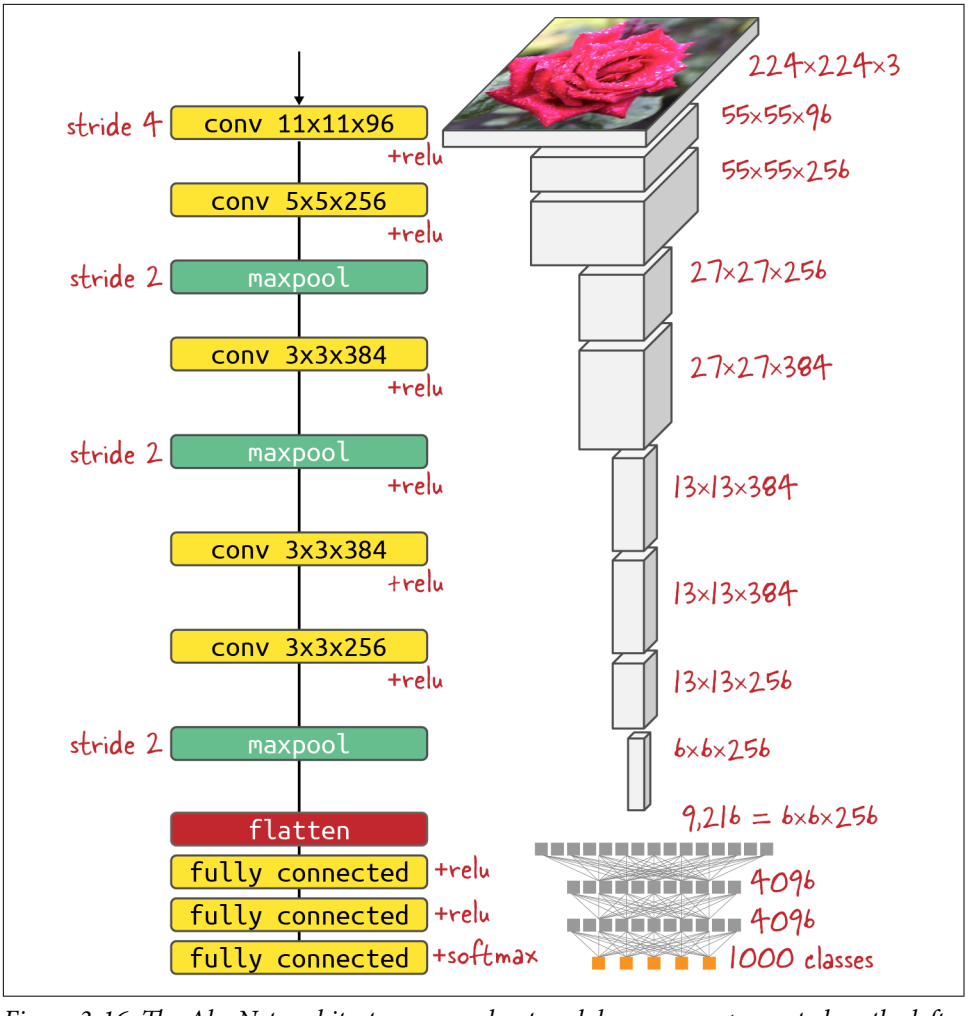


Figure 3-16. The AlexNet architecture: neural network layers are represented on the left. Feature maps (as transformed by the layers) on the right.

In this architecture, convolutional layers change the depth of the data—i.e., the number of channels. Max-pooling layers downsample the data in the height and width directions. The first convolutional layer has a stride of 4, which is why it downsamples the image as well.

AlexNet uses 3x3 max-pooling operations with a stride of 2. A more traditional choice would be 2x2 max pooling with a stride of 2. The AlexNet study claims some advantage for this "overlapping" max pooling, but it does not appear to be significant.

Every convolutional layer is activated by a ReLU activation function. The final four layers form the classification head of AlexNet, taking the last feature map, flattening

all of its values into a vector, and feeding it through three fully connected layers. Because AlexNet was designed for a thousand categories, the last layer is activated by a softmax with one thousand outputs that computes the probabilities of the thousand target classes.

All convolutional and fully connected layers use an additive bias. When the ReLU activation function is used, it is customary to initialize the bias to a small positive value before training so that, after activation, all layers start with a nonzero output and a nonzero gradient (remember that the ReLU curve is a flat zero for all negative values).

In Figure 3-16, notice that AlexNet starts with a very large 11x11 convolutional filter. This is costly in terms of learnable weights and probably not something that would be done in more modern architectures. However, one advantage of the large 11x11 filters is that their learned weights can be visualized as 11x11-pixel images. The authors of the AlexNet paper did so; their results are shown in Figure 3-17.

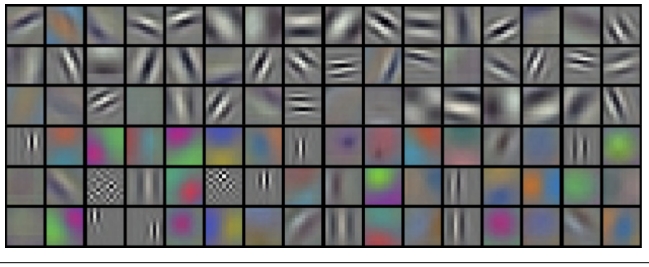


Figure 3-17. All 96 filters from the first AlexNet layer. Their size is 11x11x3, which means they can be visualized as color images. This picture shows their weights after training. Image from Krizhevsky et al., 2012.

As you can see, the network learned to detect vertical, horizontal and slanted lines of various orientations. Two filters exhibit a checkerboard pattern, which probably reacts to grainy textures in the image. You can also see detectors for single colors or pairs of adjacent colors. All these are basic features that subsequent convolutional layers will assemble into semantically more significant constructs. For example, the neural network will combine textures and lines into shapes like "wheels," "handlebars," and "saddle," and then combine these shapes into a "bicycle."

We chose to present AlexNet here because it was one of the pioneering convolutional architectures. Alternating convolutional and max-pooling layers is still a feature of modern networks. Other choices made in this architecture, however, no longer represent currently recognized best practice. For example, the use of a very large 11x11 filter in the first convolutional layer has since been found to not be the best use of learnable weights (3x3 is better, as we'll see later in this chapter). Also, the three final fully connected layers have more than 26 million learnable weights! This is an order

of magnitude more than all the convolutional layers combined (3.7 million). The network is also very shallow, with only eight neural layers. Modern neural networks increase that dramatically, to one hundred layers or more.

One advantage of this very simple model, however, is that it can be implemented quite concisely in Keras (you can see the full example in *03c_fromzero_ALEXNET_flowers104.ipynb* on GitHub):

```
model = tf.keras.Sequential([
    tf.keras.Input(shape=[IMG_HEIGHT, IMG_WIDTH, 3]),
    tf.keras.layers.Conv2D(filters=96, kernel_size=11, strides=4,
                           activation='relu'),
    tf.keras.layers.Conv2D(filters=256, kernel_size=5,
                           activation='relu'),
    tf.keras.layers.MaxPool2D(pool_size=2, strides=2),
    tf.keras.layers.Conv2D(filters=384, kernel_size=3,
                           activation='relu'),
    tf.keras.layers.MaxPool2D(pool_size=2, strides=2),
    tf.keras.layers.Conv2D(filters=384, kernel_size=3,
                           activation='relu'),
    tf.keras.layers.Conv2D(filters=256, kernel_size=3,
                           activation='relu'),
    tf.keras.layers.MaxPool2D(pool_size=2, strides=2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(4096, activation='relu'),
    tf.keras.layers.Dense(4096, activation='relu'),
    tf.keras.layers.Dense(len(CLASSES), activation='softmax')
])
```

This model converges on the 104 flowers dataset to an accuracy of 39%, which, while not useful for practical flower recognition, is surprisingly good for such a simple architecture.

AlexNet at a Glance

Architecture
: Alternates convolutional and max-pooling layers

Publication
: Alex Krizhevsky et al.,"ImageNet Classification with Deep Convolutional Neural Networks," NIPS 2012, *https://oreil.ly/X3xRb*.

Code sample
: *03c_fromzero_ALEXNET_flowers104.ipynb*

Table 3-2. AlexNet at a glance

Model	Parameters (excl. classification head[a])	ImageNet accuracy	104 flowers F1 score[b] (trained from scratch)
AlexNet	3.7M	60%	39% precision: 44%, recall: 38%

[a] Excluding classification head from parameter counts for easier comparisons between architectures. Without the classification head, the number of parameters in the network is resolution-independent. Also, in fine-tuning examples, a different classification head might be used.
[b] For accuracy, precision, recall, and F1 score values, higher is better.

In the remainder of this chapter, we provide intuitive explanations of different network architectures as well as the concepts and building blocks they introduced. Although we showed you the implementation of AlexNet in Keras, you would not typically implement the architectures that we discussed by yourself. Instead, these models are often available directly in Keras as pretrained models ready for transfer learning or fine-tuning. For example, this is how you can instantiate a pretrained ResNet50 model (for more information, see "Pretrained Models in Keras" on page 59):

```
tf.keras.applications.ResNet50(weights='imagenet')
```

If a model is not yet available in `keras.applications`, it can usually be found in TensorFlow Hub. For example, this is how you instantiate the same ResNet50 model from TensorFlow Hub:

```
hub.KerasLayer(
    "https://tfhub.dev/tensorflow/resnet_50/classification/1")
```

So, feel free to skim the rest of this chapter to get an idea of the basic concepts, and then read the final section on how to choose a model architecture for your problem. You don't need to understand all the nuances of the network architectures in this chapter to make sense of the remainder of this book, because it is rare that you will have to implement any of these architectures from scratch or design your own network architecture. Mostly, you will pick one of the architectures we suggest in the final section of this chapter. It is, however, interesting to understand how these architectures are constructed. Understanding the architectures will also help you pick the correct parameters when you instantiate them.

The Quest for Depth

After AlexNet, researchers started increasing the depths of their convolutional networks. They found that adding more layers resulted in better classification accuracy. Several explanations have been offered for this:

The expressivity argument
: A single layer is a linear function. It cannot approximate complex nonlinear functions, whatever its number of parameters. Each layer is, however, activated with a nonlinear activation function such as sigmoid or ReLU. Stacking multiple layers results in multiple successive nonlinearities and a better chance of being able to approximate the desired highly complex functionality, such as differentiating between images of cats and dogs.

The generalization argument
: Adding parameters to a single layer increases the "memory" of the neural network and allows it to learn more complex things. However, it will tend to learn them by memorizing input examples. This does not generalize well. On the other hand, stacking many layers forces the network to break down its input semantically into a hierarchical structure of features. For example, initial layers will recognize fur and whiskers, and later layers will assemble them to recognize a cat head, then an entire cat. The resulting classifier generalizes better.

The perceptive field argument
: If a cat's head covers a significant portion of an image—say, a 128x128-pixel region—a single-layer convolutional network would need 128x128 filters to be able to capture it, which would be prohibitively expensive in term of learnable weights. Stacked layers, on the other hand, can use small 3x3 or 5x5 filters and still be able to "see" any 128x128-pixel region if they are sufficiently deep in the convolutional stack.

In order to design deeper convolutional networks without growing the parameter count uncontrollably, researchers also started designing cheaper convolutional layers. Let's see how.

Filter Factorization

Which one is better: a 5x5 convolutional filter or two 3x3 filters applied in sequence? Both have a receptive area of 5x5 (see Figure 3-18). Although they do not perform the exact same mathematical operation, their effect is likely to be similar. The difference is that two 3x3 filters applied in sequence have a total of 2 * 3 * 3 = 18 learnable parameters, whereas a single 5x5 filter has 5 * 5 = 25 learnable weights. So, two 3x3 filters are cheaper.

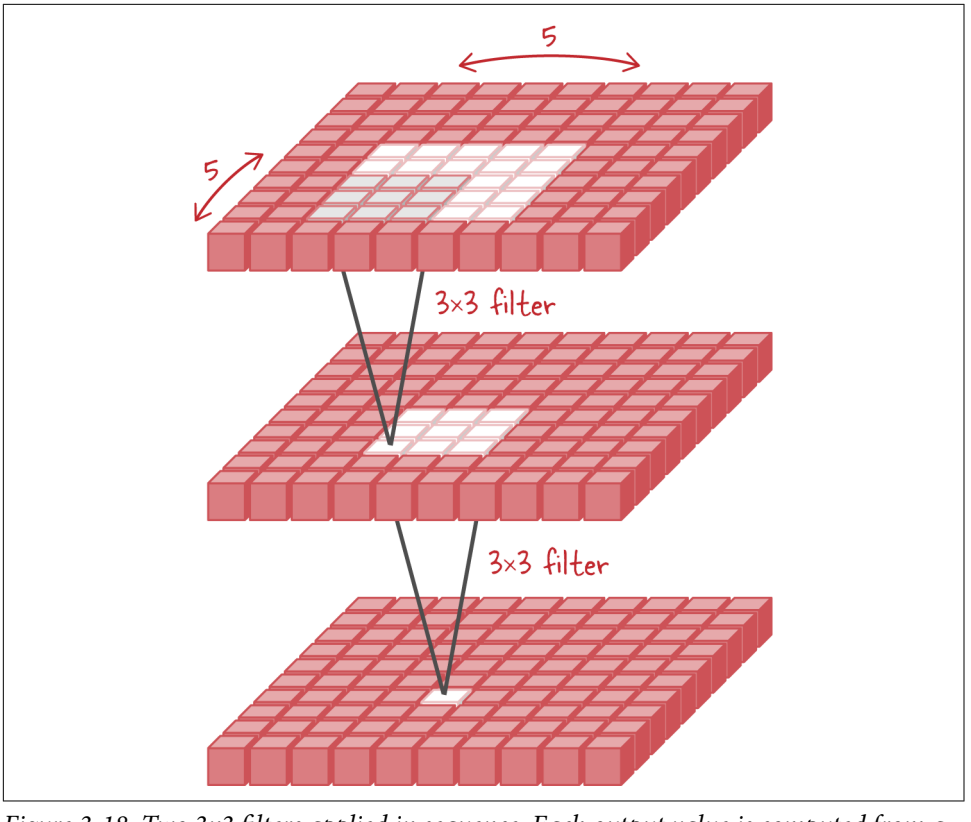


Figure 3-18. Two 3x3 filters applied in sequence. Each output value is computed from a 5x5 receptive field, which is similar to how a 5x5 filter works.

Another advantage is that a pair of 3x3 convolutional layers will involve two applications of the activation function, since each convolutional layer is followed by an activation. A single 5x5 layer has a single activation. The activation function is the only nonlinear part of a neural network and it is probable that the composition of nonlinearities in sequence will be able to express more complex nonlinear representations of the inputs.

In practice, it has been found that two 3x3 layers work better than one 5x5 layer while using fewer learnable weights. That's why you will see 3x3 convolutional layers used extensively in modern convolutional architectures. This is sometimes referred to as *filter factorization*, although it is not exactly a factorization in the mathematical sense.

The other filter size that is popular today is 1x1 convolutions. Let's see why.

1x1 Convolutions

Sliding a single-pixel filter across an image sounds silly. It's multiplying the image by a constant. However, on multichannel inputs, with a different weight for each channel, it actually makes sense. For example, multiplying the three color channels of an RGB image by three learnable weights and then adding them up produces a linear combination of the color channels that can actually be useful. A 1x1 convolutional layer performs multiple linear combinations of this kind, each time with an independent set of weights, producing multiple output channels (Figure 3-19).

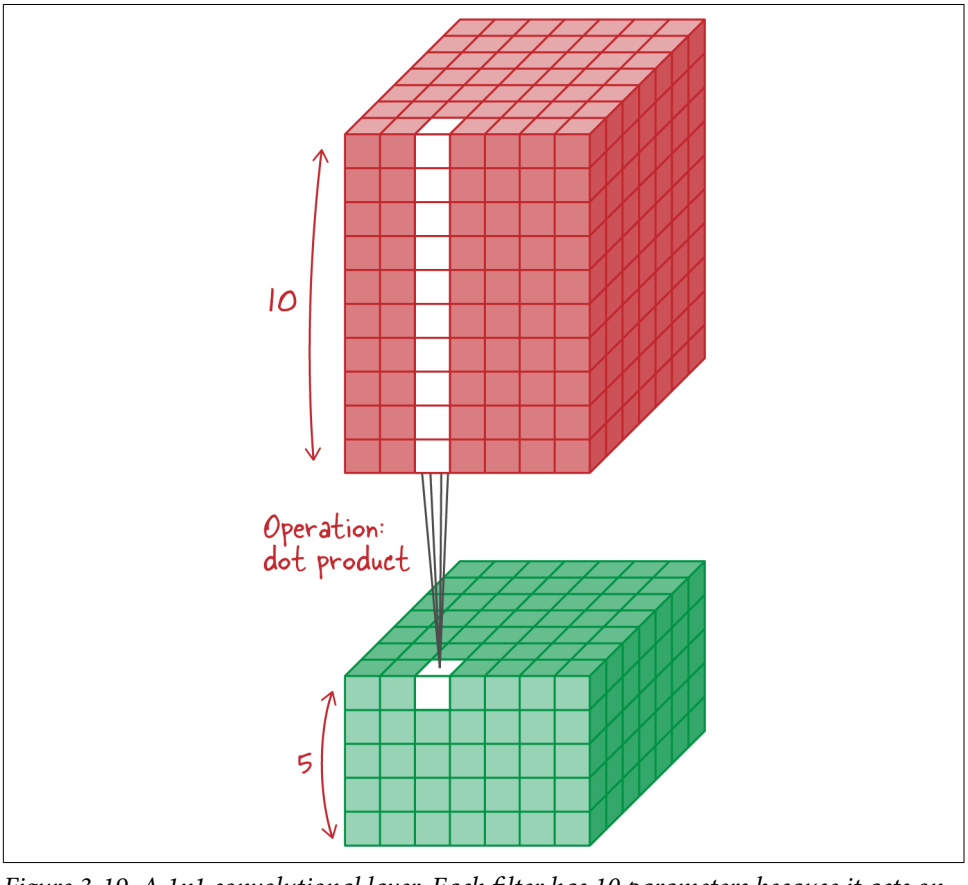


Figure 3-19. A 1x1 convolutional layer. Each filter has 10 parameters because it acts on a 10-channel input. 5 such filters are applied, each with its own learnable parameters (not shown in the figure), resulting in 5 channels of output data.

A 1x1 convolutional layer is a useful tool for adjusting the number of channels of the data. The second advantage is that 1x1 convolutional layers are cheap, in terms of number of learnable parameters, compared to 2x2, 3x3, or larger layers. The tensor of

weights representing the 1x1 convolutional layer in the previous illustration is shown in Figure 3-20.

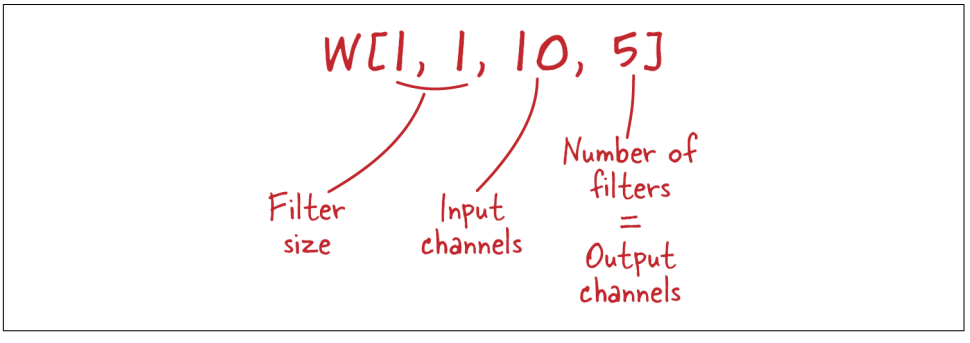


Figure 3-20. The weights matrix of the 1x1 convolutional layer from Figure 3-19.

The number of learnable weights is 1 * 1 * 10 * 5 = 50. A 3x3 layer with the same number of input and output channels would require 3 * 3 * 10 * 5 = 450 weights, an order of magnitude more!

Next, let's look at an architecture that employs these tricks.

VGG19

VGG19, introduced in a 2014 paper by Karen Simonyan and Andrew Zisserman, was one of the first architectures to use 3x3 convolutions exclusively. Figure 3-21 shows what it looks like with 19 layers.

All the neural network layers in this figure use biases and are ReLU-activated, apart from the last layer which uses softmax activation.

VGG19 improves on AlexNet by being much deeper. It has 16 convolutional layers instead of 5. It also uses 3x3 convolutions exclusively without losing accuracy. However, it uses the exact same classification head as AlexNet, with three large fully connected layers accounting for over 120 million weights, while it has only 20 million weights in the convolutional layers. There are cheaper alternatives.

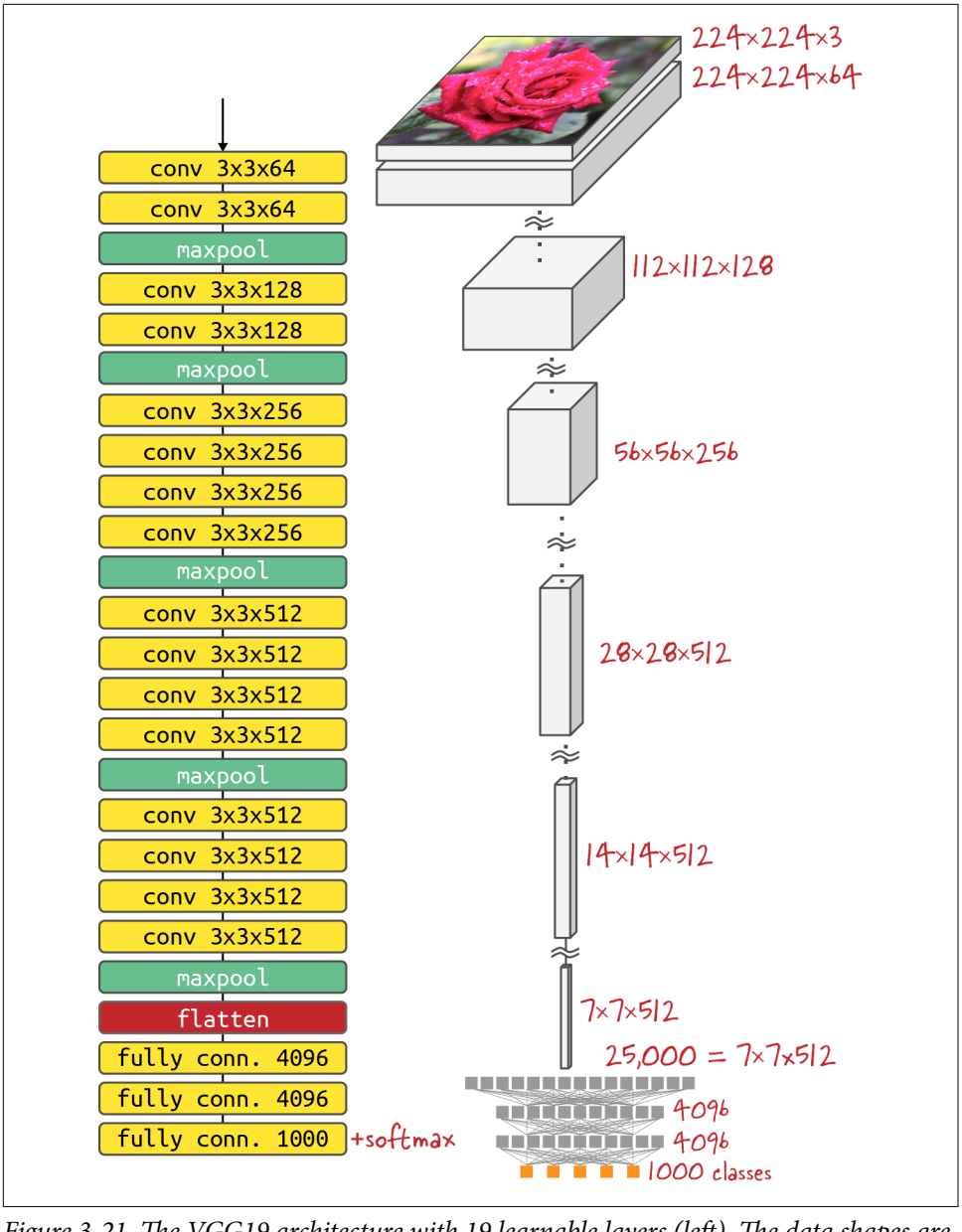


Figure 3-21. The VGG19 architecture with 19 learnable layers (left). The data shapes are shown on the right (not all represented). Notice that all convolutional layers use 3x3 filters.

VGG19 at a Glance

Architecture
: Alternates convolutional and max-pooling layers using 3x3 convolutions only

Publication
: Karen Simonyan and Andrew Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," 2014, *https://arxiv.org/abs/1409.1556v6*.

Code sample
: *03d_finetune_VGG19_flowers104.ipynb*

Table 3-3. VGG19 at a glance

Model	Parameters (excl. classif. head[a])	ImageNet accuracy	104 flowers F1 score[b] (fine-tuning)	104 flowers F1 score (trained from scratch)
VGG19	20M	71%	88% precision: 89%, recall: 88%	N/A[c]
Previous best for comparison:				
AlexNet	3.7M	60%		39% precision: 44%, recall: 38%

[a] Excluding classification head from parameter counts for easier comparisons between architectures. Without the classification head, the number of parameters in the network is resolution-independent. Also, in fine-tuning examples, a different classification head might be used.
[b] For accuracy, precision, recall, and F1 score values, higher is better.
[c] We did not bother training VGG16 from scratch on the 104 flowers dataset because the result would be much worse than fine-tuning.

Global Average Pooling

Let's look again at the implementation of the classification head. In both the AlexNet and VGG19 architectures, the feature map output by the last convolutional layer is turned into a vector (flattened) and then fed into one or more fully connected layers (see Figure 3-22). The goal is to end on a softmax-activated fully connected layer with exactly as many neurons as classes in the classification problem at hand—for example, one thousand classes for the ImageNet dataset or five classes for the 5-flowers dataset used in the previous chapter. This fully connected layer has input * outputs weights, which tends to be a lot.

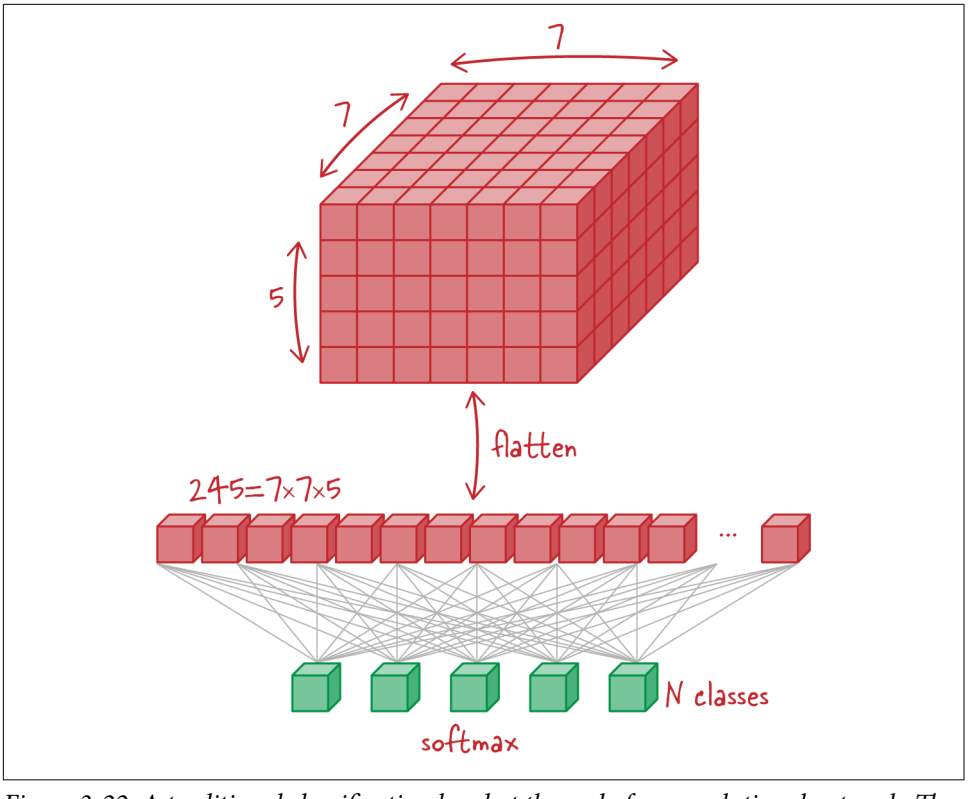


Figure 3-22. A traditional classification head at the end of a convolutional network. The data coming out of convolutional layers is flattened and fed into a fully connected layer. Softmax activation is used to obtain class probabilities.

If the only goal is to obtain N values to feed an N-way softmax function, there is an easy way to achieve that: adjust the convolutional stack so that it ends on a final feature map with exactly N channels and simply average the values in each channel, as shown in Figure 3-23. This is known as *global average pooling*. Global average pooling involves no learnable weights, so from this perspective it's cheap.

Global average pooling can be followed by a softmax activation directly (as in SqueezeNet, shown in Figure 3-26), although in most architectures described in this book it will be followed by a single softmax-activated fully connected layer (for example in a ResNet, as shown in Figure 3-29).

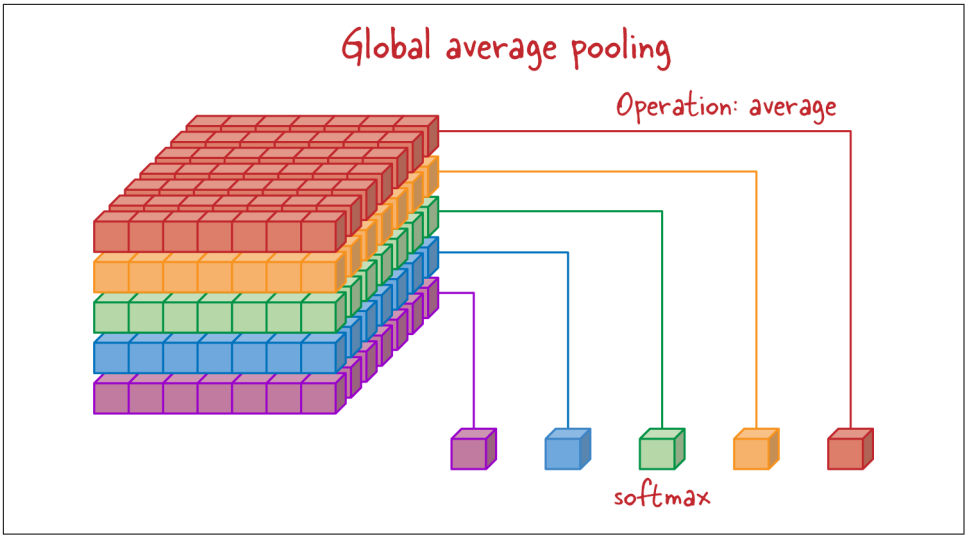


Figure 3-23. Global average pooling. Each channel is averaged into a single value. Global average pooling followed by a softmax function implements a classification head with zero learnable parameters.

Averaging removes a lot of the positional information present in the channels. That might or might not be a good thing depending on the application. Convolutional filters detect the things they have been trained to detect in a specific location. If the network is classifying, for example, cats versus dogs, the location data (e.g., "cat's whiskers" detected at position *x, y* in the channel) is probably not useful in the classification head. The only thing of interest is the "dog detected anywhere" signal versus the "cat detected anywhere" signal. For other applications, though, a global average-pooling layer might not be the best choice. For example, in object detection or object counting use cases, the location of detected objects is important and global average pooling should not be used.

Modular Architectures

A straight succession of convolutional and pooling layers is enough to build a basic convolutional neural network. However, to further increase prediction accuracy, researchers designed more complex building blocks, or *modules*, often given arcane names such as "Inception modules," "residual blocks," or "inverted residual bottlenecks," and then assembled them into complete convolutional architectures. Having higher-level building blocks also made it easier to create automated algorithms to search for better architectures, as we will see in the section on neural architecture

search. In this section we'll explore several of these modular architectures and the research behind each of them.

Inception

The Inception architecture was named after Christopher Nolan's 2010 movie *Inception*, starring Leonardo DiCaprio. One line from the movie dialog—"We need to go deeper"—became an internet meme. Building deeper and deeper neural networks was one of the main motivations of researchers at that time.

The Inception V3 architecture uses 3x3 and 1x1 convolutional filters exclusively, as is now customary in most convolutional architectures. It tries to address another problem, though, with a very original approach. When lining up the convolutional and pooling layers in a neural network, the designer has multiple choices, and the best one is not obvious. Instead of relying on guesswork and experimentation, why not build multiple options into the network itself and let it learn which one is the best? This is the motivation behind Inception's "modules" (see Figure 3-24).

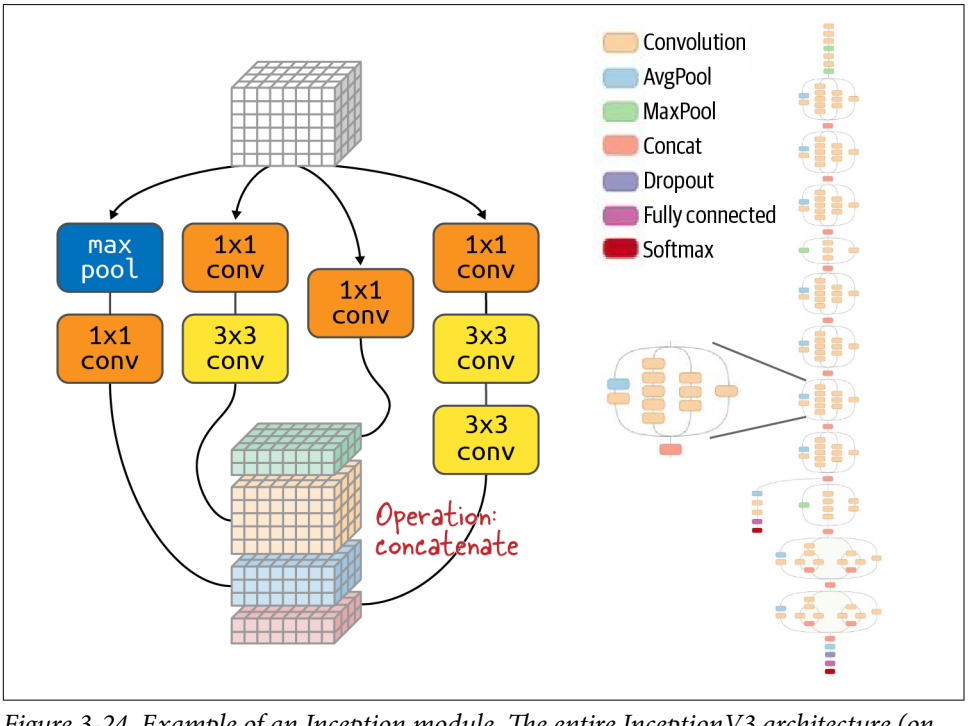


Figure 3-24. Example of an Inception module. The entire InceptionV3 architecture (on the right) is made up of many such modules.

Instead of deciding beforehand which sequence of layers is the most appropriate, an Inception module provides several alternatives that the network can choose from,

based on data and training. As shown in Figure 3-24, the outputs of the different paths are concatenated into the final feature map.

We will not detail the full InceptionV3 architecture in this book because it is rather complex and has since been superseded by newer and simpler alternatives. A simplified variant, also based on the "module" idea, is presented next.

InceptionV3 at a Glance

Architecture
: Sequence of multipath convolutional modules.

Publication
: Christian Szegedy et al., "Rethinking the Inception Architecture for Computer Vision," 2015, *https://arxiv.org/abs/1512.00567v3*.

Code sample
: *03e_finetune_INCEPTIONV3_flowers104.ipynb*

Table 3-4. InceptionV3 at a glance

Model	Parameters (excl. classification head[a])	ImageNet accuracy	104 flowers F1 score[b] (fine-tuning)
InceptionV3	22M	78%	95% precision: 95%, recall: 94%
Previous best for comparison:			
VGG19	20M	71%	88% precision: 89%, recall: 88%

[a] Excluding classification head from parameter counts for easier comparisons between architectures. Without the classification head, the number of parameters in the network is resolution-independent. Also, in fine-tuning examples, a different classification head might be used.
[b] For accuracy, precision, recall, and F1 score values, higher is better.

SqueezeNet

The idea of modules was simplified by the SqueezeNet architecture, which kept the basic principle of offering multiple paths for the network to choose from but streamlined the modules themselves into their simplest expression (Figure 3-25). The SqueezeNet paper calls them "fire modules."

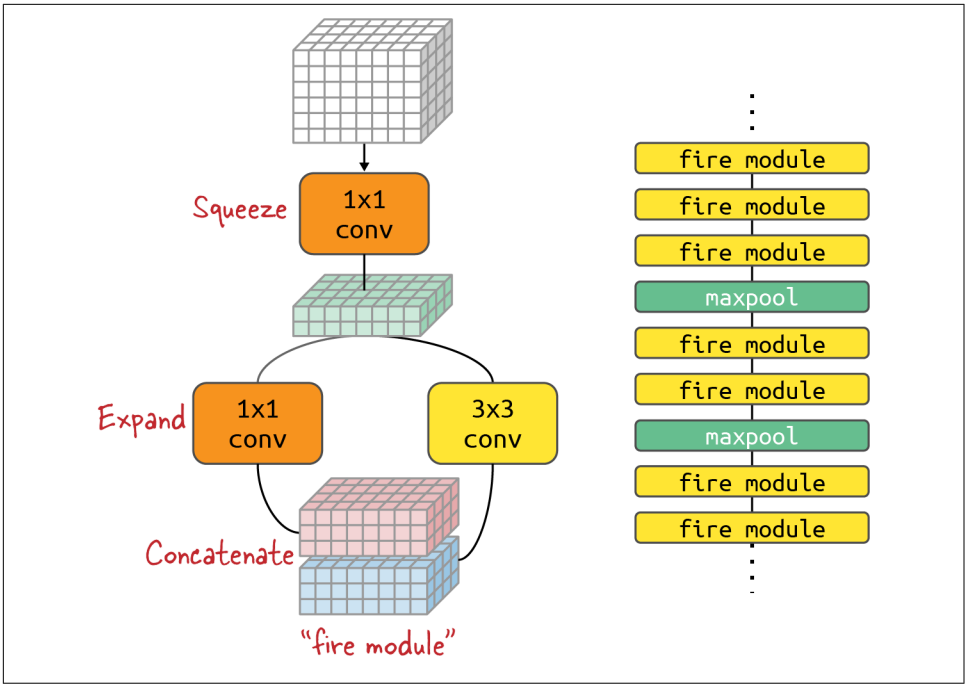


Figure 3-25. A simplified and standardized convolutional module from the SqueezeNet architecture. The architecture, shown on the right, alternates these "fire modules" with max-pooling layers.

The modules used in the SqueezeNet architecture alternate a contraction stage, where the number of channels is reduced by a 1x1 convolution, with an expansion stage where the number of channels is increased again.

To save on weight count, SqueezeNet uses global average pooling for the last layer. Also, two out of the three convolutional layers in each module are 1x1 convolutions, which saves on learnable weights (see Figure 3-26).

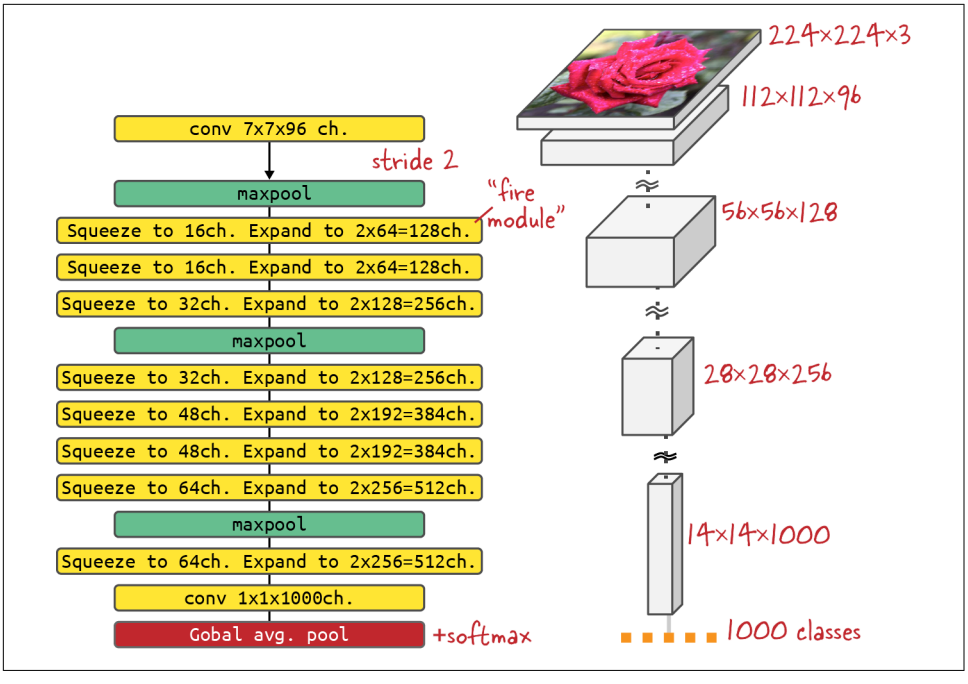


Figure 3-26. The SqueezeNet architecture with 18 convolutional layers. Each "fire module" contains a "squeeze" layer followed by two parallel "expand" layers. The network pictured here contains 1.2M learnable parameters.

In Figure 3-26, "maxpool" is a standard 2x2 max-pooling operation with a stride of 2. Also, every convolutional layer in the architecture is ReLU-activated and uses batch normalization. The thousand-class classification head is implemented by first stretching the number of channels to one thousand with a 1x1 convolution, then averaging the thousand channels (global average pooling) and finally applying softmax activation.

The SqueezeNet architecture aims to be simple and economical (in terms of learnable weights) but still incorporate most of the latest best practices in building convolutional neural networks. Its simplicity makes it a good choice when you want to implement your own convolutional backbone, either for education purposes or because you need to tweak it for your needs. The one architectural element that might not be considered best practice today is the large 7x7 initial convolutional layer, inspired directly by AlexNet.

In order to implement the SqueezeNet model in Keras, we have to use a Keras Functional API model. We can no longer use a Sequential model, because SqueezeNet is not a straight sequence of layers. We first create a helper function that instantiates a

fire module (the full code is available in *03f_fromzero_SQUEEZENET24_flowers104.ipynb* on GitHub):

```
def fire(x, squeeze, expand):
    y  = tf.keras.layers.Conv2D(filters=squeeze, kernel_size=1,
                                activation='relu', padding='same')(x)
    y = tf.keras.layers.BatchNormalization()(y)
    y1 = tf.keras.layers.Conv2D(filters=expand//2, kernel_size=1,
                                activation='relu', padding='same')(y)
    y1 = tf.keras.layers.BatchNormalization()(y1)
    y3 = tf.keras.layers.Conv2D(filters=expand//2, kernel_size=3,
                                activation='relu', padding='same')(y)
    y3 = tf.keras.layers.BatchNormalization()(y3)
    return tf.keras.layers.concatenate([y1, y3])
```

As you can see in the first line of the function, using the Keras Functional API, `tf.keras.layers.Conv2D()` instantiates a convolutional layer which is then called with the input `x`. We can slightly transform the `fire()` function so that it uses the same semantics:

```
def fire_module(squeeze, expand):
    return lambda x: fire(x, squeeze, expand)
```

And here is the implementation of a custom 24-layer SqueezeNet. It performed reasonably well on the 104 flowers dataset, with an F1 score of 76%, which isn't bad considering it was trained from scratch:

```
x = tf.keras.layers.Input(shape=[IMG_HEIGHT, IMG_WIDTH, 3])
y = tf.keras.layers.Conv2D(kernel_size=3, filters=32,
                           padding='same', activation='relu')(x)
y = tf.keras.layers.BatchNormalization()(y)
y = fire_module(16, 32)(y)
y = tf.keras.layers.MaxPooling2D(pool_size=2)(y)
y = fire_module(48, 96)(y)
y = tf.keras.layers.MaxPooling2D(pool_size=2)(y)
y = fire_module(64, 128)(y)
y = fire_module(80, 160)(y)
y = fire_module(96, 192)(y)
y = tf.keras.layers.MaxPooling2D(pool_size=2)(y)
y = fire_module(112, 224)(y)
y = fire_module(128, 256)(y)
y = fire_module(160, 320)(y)
y = tf.keras.layers.MaxPooling2D(pool_size=2)(y)
y = fire_module(192, 384)(y)
y = fire_module(224, 448)(y)
y = tf.keras.layers.MaxPooling2D(pool_size=2)(y)
y = fire_module(256, 512)(y)
y = tf.keras.layers.GlobalAveragePooling2D()(y)
y = tf.keras.layers.Dense(len(CLASSES), activation='softmax')(y)

model = tf.keras.Model(x, y)
```

In the last line, we create the model by passing in the initial input layer and the final output. The model can be used just like a Sequential model, so the rest of the code remains the same.

SqueezeNet at a Glance

Architecture
Simplified convolutional modules built from parallel 3x3 and 1x1 convolutions

Publication
Forrest Iandola et al., "SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters," 2016, *https://arxiv.org/abs/1602.07360*.

Code sample
03f_fromzero_SQUEEZENET24_flowers104.ipynb

Table 3-5. SqueezeNet at a glance

Model	Parameters (excl. classification head[a])	ImageNet accuracy	104 flowers F1 score[b] (trained from scratch)
SqueezeNet, 24 layers	2.7M		76% precision: 77%, recall: 75%
SqueezeNet, 18 layers	1.2M	56%	
Previous best for comparison:			
AlexNet	3.7M	60%	39% prec.: 44%, recall: 38%

[a] Excluding classification head from parameter counts for easier comparisons between architectures. Without the classification head, the number of parameters in the network is resolution-independent. Also, in fine-tuning examples, a different classification head might be used.
[b] For accuracy, precision, recall, and F1 score values, higher is better.

ResNet and Skip Connections

The ResNet architecture, introduced in a 2015 paper by Kaiming He et al., continued the trend of increasing the depth of neural networks but addressed a common challenge with very deep neural networks—they tend to converge badly because of vanishing or exploding gradient problems. During training, a neural network sees what error (or loss) it is making and tries to minimize this error by adjusting its internal weights. It is guided in this by the first derivative (or gradient) of the error. Unfortunately, with many layers, the gradients tend to be spread too thin across all layers and the network converges slowly or not at all.

ResNet tried to remedy this by adding *skip connections* alongside its convolutional layers (Figure 3-27). Skip connections convey the signal as is, then recombine it with the data that has been transformed by one or more convolutional layers. The combining operation is a simple element-by-element addition.

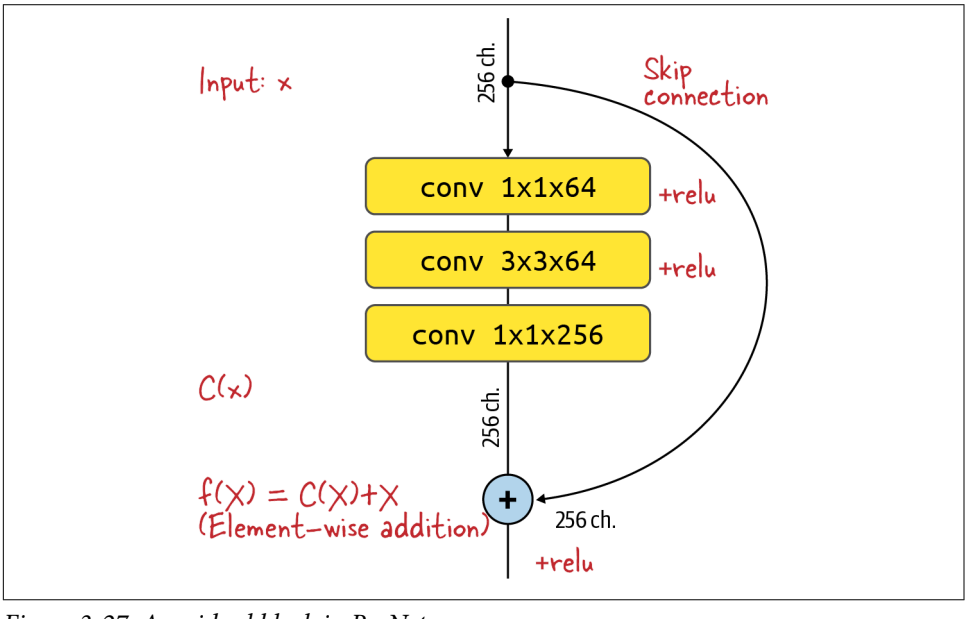


Figure 3-27. A residual block in ResNet.

As can be seen in Figure 3-27, the output of the block $f(x)$ is the sum of the output of the convolutional path $C(x)$ and the skip connection (x). The convolutional path is trained to compute $C(x) = f(x) - x$, the difference between the desired output and the input. The authors of the ResNet paper argue that this "residue" is easier for the network to learn.

An obvious limitation is that the element-wise addition can only work if the dimensions of the data remain unchanged. The sequence of layers that is straddled by a skip connection (called a *residual block*) must preserve the height, the width, and the number of channels of the data.

When size adjustments are needed, a different kind of residual block is used (Figure 3-28). Different numbers of channels can be matched in the skip connection by using a 1x1 convolution instead of an identity. Height and width adjustments are obtained by using a stride of 2 both in the convolutional path and in the skip connection (yes, implementing the skip connection with a 1x1 convolution of stride 2 ignores half of the values in the input, but this does not seem to matter in practice).

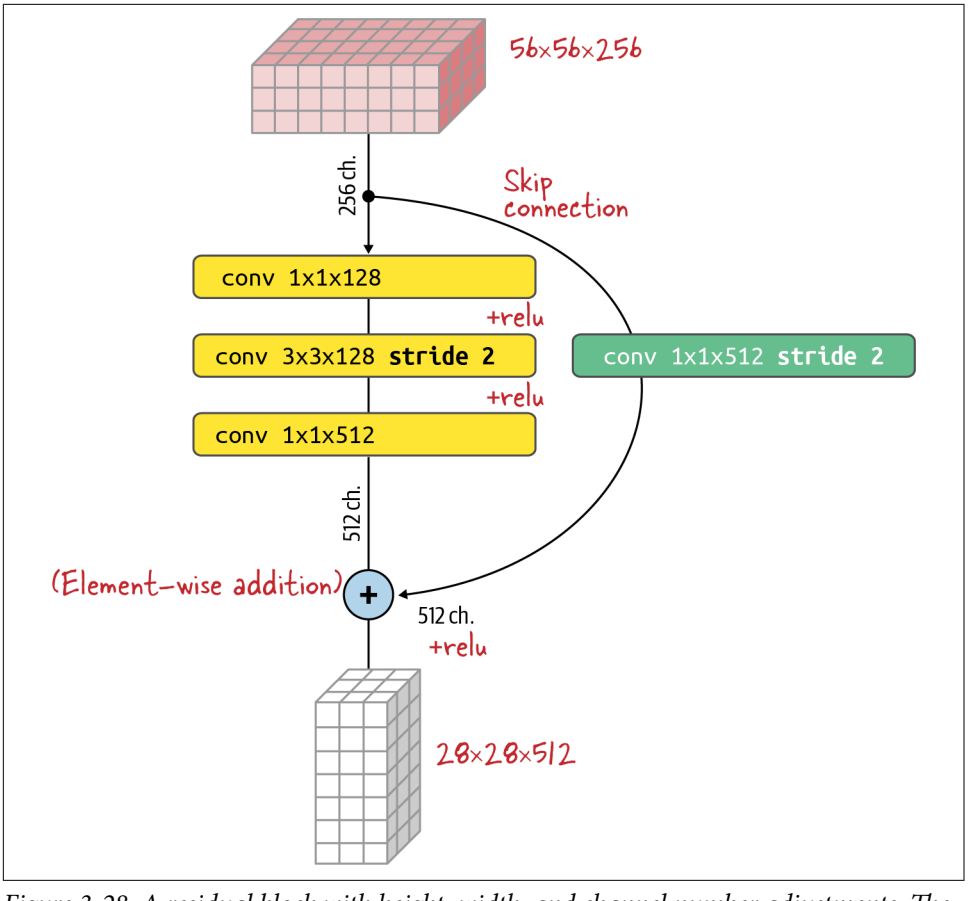


Figure 3-28. A residual block with height, width, and channel number adjustments. The number of channels is changed in the skip connection by using a 1x1 convolution instead of an identity function. Data height and width are downsampled by using one convolutional layer with a stride of 2 in both the convolutional path and the skip connection.

The ResNet architecture can be instantiated with various depths by stacking more and more residual blocks. Popular sizes are ResNet50 and ResNet101 (Figure 3-29).

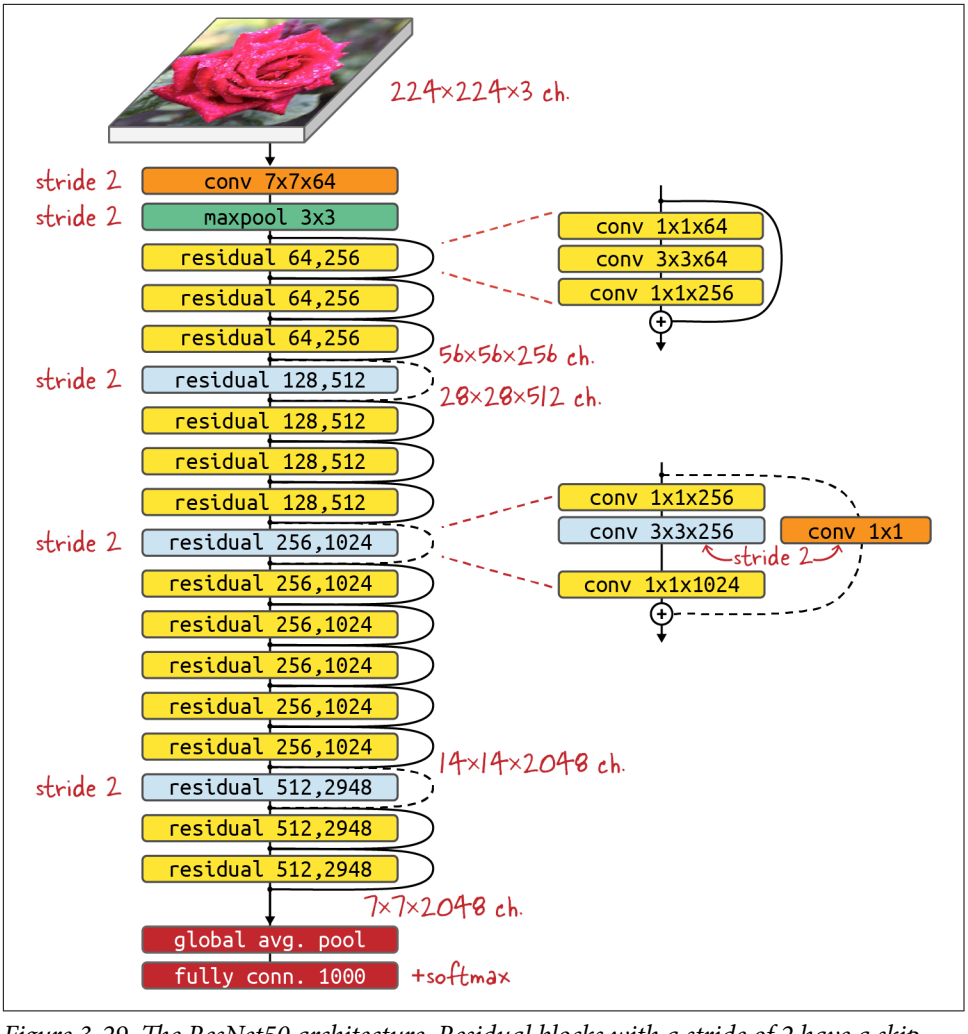


Figure 3-29. The ResNet50 architecture. Residual blocks with a stride of 2 have a skip connection implemented using a 1x1 convolution (dotted line). The ResNet 101 architecture is similar, with the "residual 256, 1,024" block repeated 23 times instead of 6.

In Figure 3-29, all the convolutional layers are ReLU-activated and use batch normalization. A network with this architecture can grow very deep—as the names indicate, 50, 100, or more layers are common for ResNets—but it is still able to figure out which layers need to have their weights adjusted for any given output error.

Skip connections seem to help gradients flow through the network during the optimization (backpropagation) phase. Several explanations have been suggested for this. Here are the three most popular ones.

The ResNet paper's authors theorize that the addition operation (see Figure 3-27) plays an important role. In a regular neural network, internal weights are adjusted to produce a desired output, such as a classification into one thousand classes. With skip connections, however, the goal of the neural network layers is to output the delta (or "residue") between the input and the desired final output. This, the authors argue, is an "easier" task for the network, but they don't elaborate on what makes it easier.

A second interesting explanation is that residual connections actually make the network shallower. During the gradient backpropagation phase, gradients flow both through convolutional layers, where they might decrease in magnitude, and through the skip connections, which leave them unchanged. In the paper "Residual Networks Behave Like Ensembles of Relatively Shallow Networks," Veit et al. measured the intensity of gradients in a ResNet architecture. The result (Figure 3-30) shows how, in a 50-layer ResNet neural network, the signal can flow through various combinations of convolutional layers and skip connections.

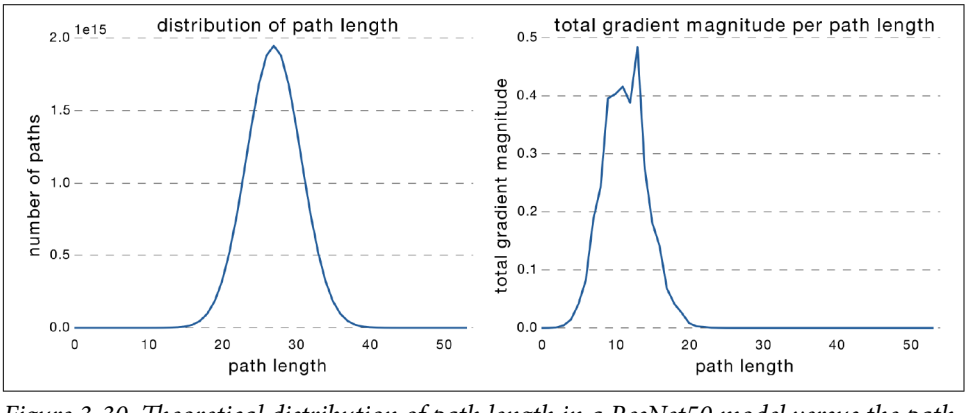


Figure 3-30. Theoretical distribution of path length in a ResNet50 model versus the path actually taken by meaningful gradients during backpropagation. Image from Veit et al., 2016.

The most likely path lengths, measured in the number of convolutional layers traversed, are midway between 0 and 50 (left graph). However, Veit et al. measured that paths providing actually useful nonzero gradients in a trained ResNet were even shorter than that, traversing around 12 layers.

According to this theory, a deep 50- or 100-layer ResNet acts as an ensemble—i.e., a collection of shallower networks that optimally solve different parts of a classification problem. Taken together, they pool their classification strengths, but they still converge efficiently because they are not actually very deep. The benefit of the ResNet architecture compared to an actual ensemble of models is that it trains as a single model and learns to select the best path for each input by itself.

A third explanation looks at the topological landscape of the loss function optimized during training. In "Visualizing the Loss Landscape of Neural Nets", Li et al. managed to picture the loss landscape in 3D rather than its original million or so dimensions and showed that good minima were much more accessible when skip connections were used (Figure 3-31).

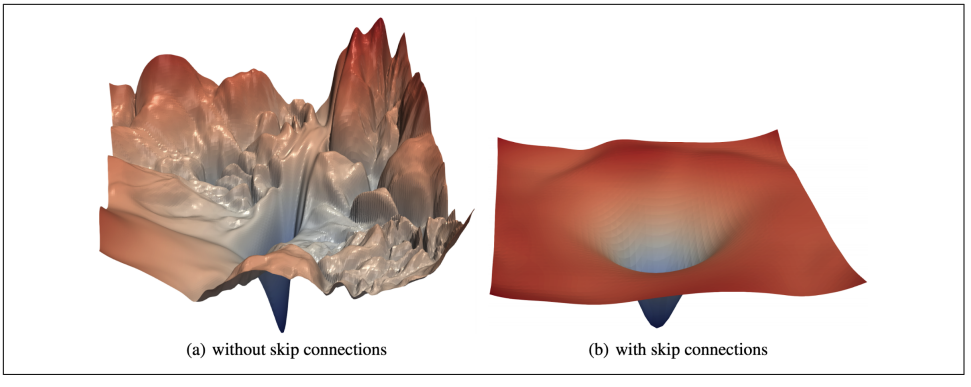


Figure 3-31. The loss landscape of a 56-layer ResNet as visualized through the "filter normalization scheme" of Li et al. Adding skip connections makes the global minimum much easier to reach. Image from Li et al., 2017.

In practice, the ResNet architecture works very well and has become one of the most popular convolutional architectures in the field, as well as the benchmark against which all other advances are measured.

ResNet at a Glance

Architecture
: Convolutional modules with skip connections

Publication
: Kaiming He et al., "Deep Residual Learning for Image Recognition," 2015, *https://arxiv.org/abs/1512.03385*.

Code samples
: *03g_finetune_RESNET50_flowers104.ipynb* and *03g_fromzero_RESNET50_flowers104.ipynb*

Table 3-6. ResNet at a glance

Model	Parameters (excl. classif. head[a])	ImageNet accuracy	104 flowers F1 score[b] (fine-tuning)	104 flowers F1 score (trained from scratch)
ResNet50	23M	75%	94% prec.: 95%, recall: 94%	73% prec.: 76%, recall: 72%
Previous best for comparison:				
InceptionV3	22M	78%	95% prec.: 95%, recall: 94%	
SqueezeNet, 24 layers	2.7M			76% prec.: 77%, recall: 75%

[a] Excluding classification head from parameter counts for easier comparisons between architectures. Without the classification head, the number of parameters in the network is resolution-independent. Also, in fine-tuning examples, a different classification head might be used.
[b] For accuracy, precision, recall, and F1 score values, higher is better.

The classification performance is a bit below that achieved by InceptionV3, but the main goal of ResNet was to allow very deep architectures to still train and converge. Beyond ResNet50, ResNet101 and ResNet152 variants are also available with 101 and 152 layers.

DenseNet

The DenseNet architecture revisits the concept of skip connections with a radical new idea. In their paper on DenseNet, Gao Huang et al. suggest feeding a convolutional layer with all the outputs of the previous layers, by creating as many skip connections as necessary. This time, data is combined by concatenation along the depth axis (channels) instead of being added, as in ResNet. Apparently, the intuition that led to the ResNet architecture—that data from skip connections should be added in because "residual" signals were easier to learn—was not fundamental. Concatenation works too.

Dense blocks are the basic building blocks of DenseNet. In a dense block, convolutions are grouped in pairs, with each pair of convolutions receiving as its input the output of all previous convolution pairs. In the dense block depicted in Figure 3-32, data is combined by concatenating it channel-wise. All convolutions are ReLU-activated and use batch normalization. Channel-wise concatenation only works if the height and width dimensions of the data are the same, so convolutions in a dense block are all of stride 1 and do not change these dimensions. Pooling layers will have to be inserted between dense blocks.

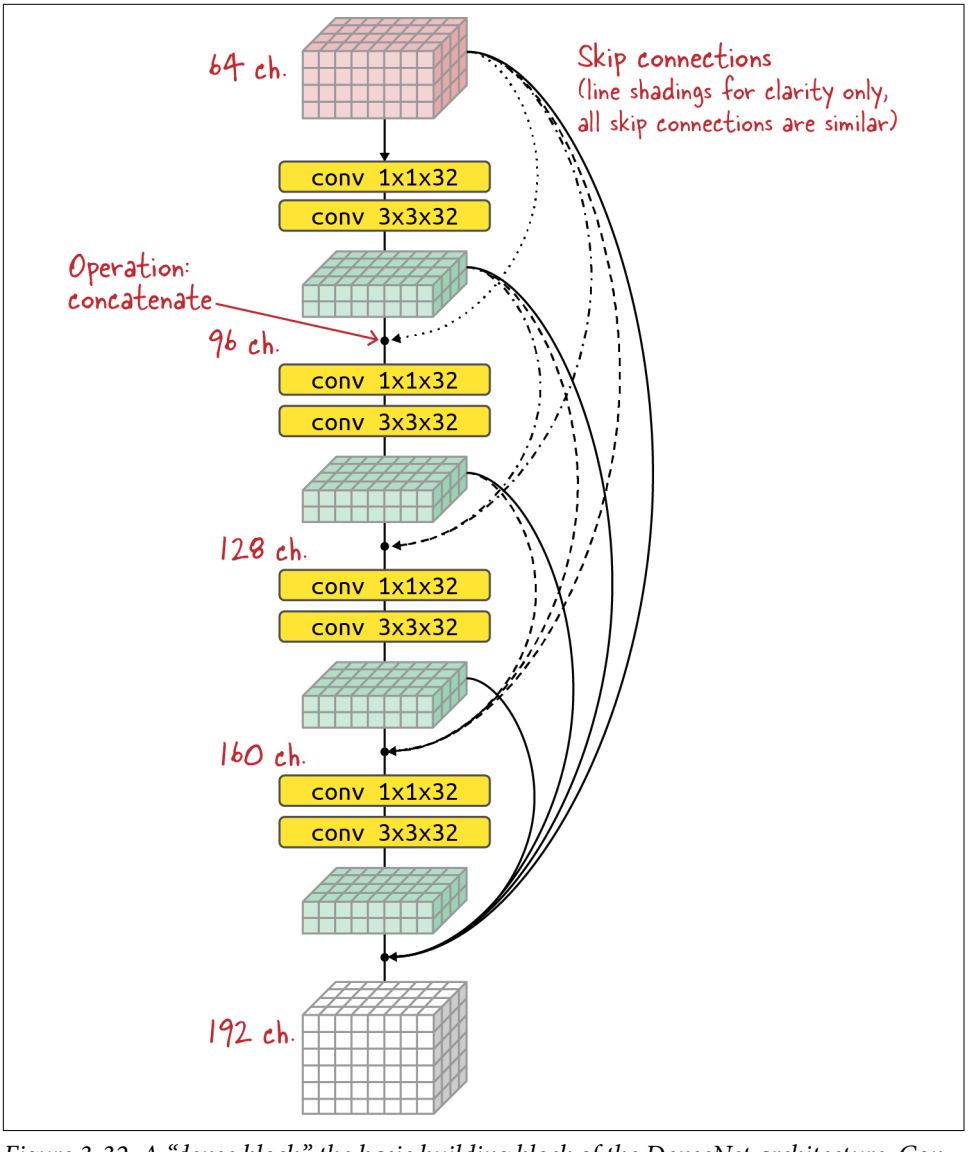


Figure 3-32. A "dense block," the basic building block of the DenseNet architecture. Convolutions are grouped in pairs. Each pair of convolutions receives as input the output of all previous convolution pairs. Notice that the number of channels grows linearly with the number of layers.

Intuitively, one would think that concatenating all previously seen outputs would lead to an explosive growth of the number of channels and parameters, but that is not in fact the case. DenseNet is surprisingly economical in terms of learnable parameters. The reason is that every concatenated block, which might have a relatively large number of channels, is always fed first through a 1x1 convolution that reduces it to a small number of channels, *K*. 1x1 convolutions are cheap in their number of parameters. A 3x3 convolution with the same number of channels (*K*) follows. The *K* resulting channels are then concatenated to the collection of all previously generated outputs. Each step, which uses a pair of 1x1 and 3x3 convolutions, adds exactly *K* channels to the data. Therefore, the number of channels grows only linearly with the number of convolutional steps in the dense block. The growth rate *K* is a constant throughout the network, and DenseNet has been shown to perform well with fairly low values of *K* (between *K*=12 and *K*=40 in the original paper).

Dense blocks and pooling layers are interleaved to create a full DenseNet network. Figure 3-33 shows a DenseNet121 with 121 layers, but the architecture is configurable and can easily scale beyond 200 layers.

The use of shallow convolutional layers (*K*=32, for example) is a characteristic feature of DenseNet. In previous architectures, convolutions with over one thousand filters were not rare. DenseNet can afford to use shallow convolutions because each convolutional layer sees all previously computed features. In other architectures, the data is transformed at each layer and the network must do active work to preserve a channel of data as-is, if that is the right thing to do. It must use some of its filter parameters to create an identity function, which is wasteful. DenseNet, the authors argue, is built to allow feature reuse and therefore requires far fewer filters per convolutional layer.

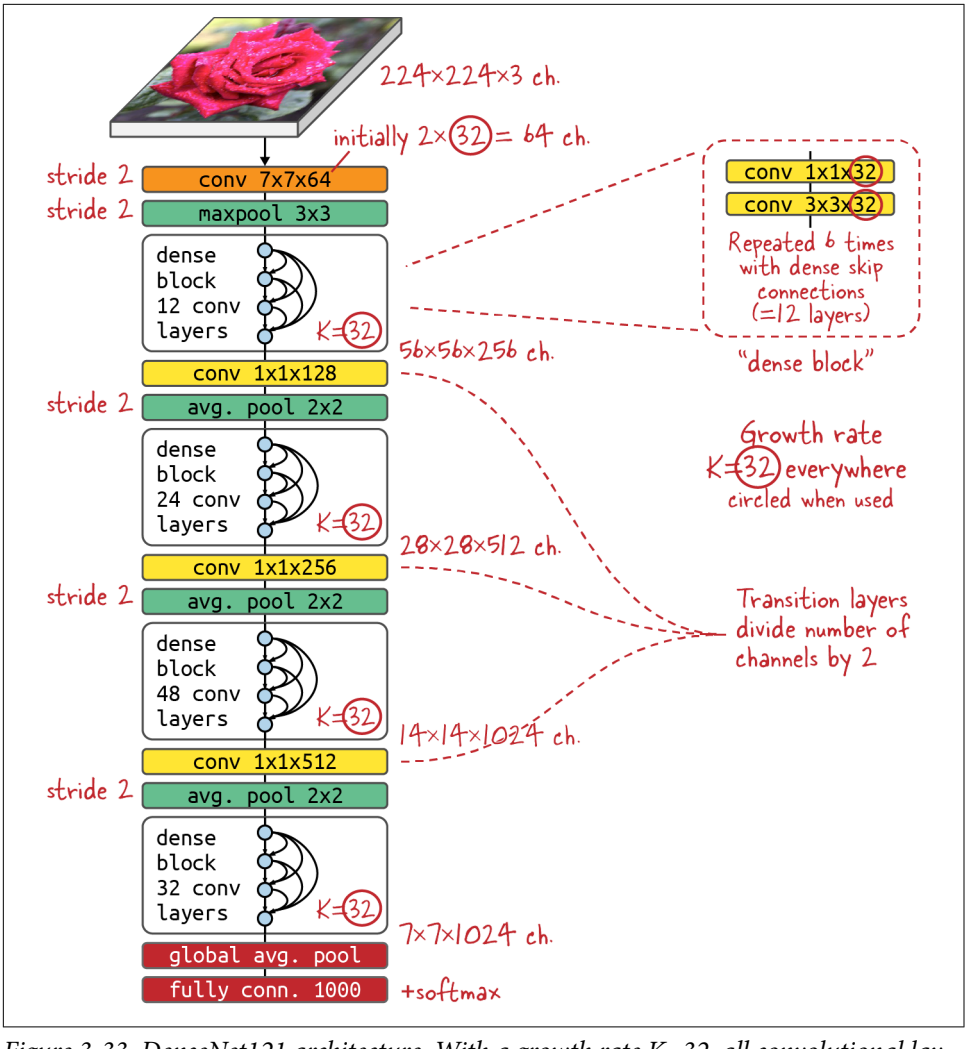


Figure 3-33. DenseNet121 architecture. With a growth rate K=32, all convolutional layers produce 32 channels of output, apart from the 1x1 convolutions used as transitions between dense blocks, which are designed to halve the number of channels. See previous figure for details about dense blocks. All convolutions are ReLU-activated and use batch normalization.

DenseNet at a Glance

Architecture
Convolutional modules with a dense network of skip connections

Publication
Gao Huang et al., "Densely Connected Convolutional Networks," 2016, *https://arxiv.org/abs/1608.06993*.

Code samples
03h_finetune_DENSENET201_flowers104.ipynb and *03h_fromzero_DENSENET121_flowers104.ipynb*

Table 3-7. DenseNet at a glance

Model	Parameters (excl. classif. head[a])	ImageNet accuracy	104 flowers F1 score[b] (fine-tuning)	104 flowers F1 score (trained from scratch)
DenseNet201	18M	77%	95% prec.: 96%, recall: 95%	
DenseNet121	7M	75%		76% prec.: 80%, recall: 74%
Previous best for comparison:				
InceptionV3	22M	78%	95% prec.: 95%, recall: 94%	
SqueezeNet, 24 layers	2.7M			76% prec.: 77%, recall: 75%

[a] Excluding classification head from parameter counts for easier comparisons between architectures. Without the classification head, the number of parameters in the network is resolution-independent. Also, in fine-tuning examples, a different classification head might be used.
[b] For accuracy, precision, recall, and F1 score values, higher is better.

Depth-Separable Convolutions

Traditional convolutions filter all the channels of the input at once. They then use many filters to give the network the opportunity to do many different things with the same input channels. Let's take the example of a 3x3 convolutional layer applied to 8 input channels with 16 output channels. It has 16 convolutional filters of shape 3x3x8 (Figure 3-34).

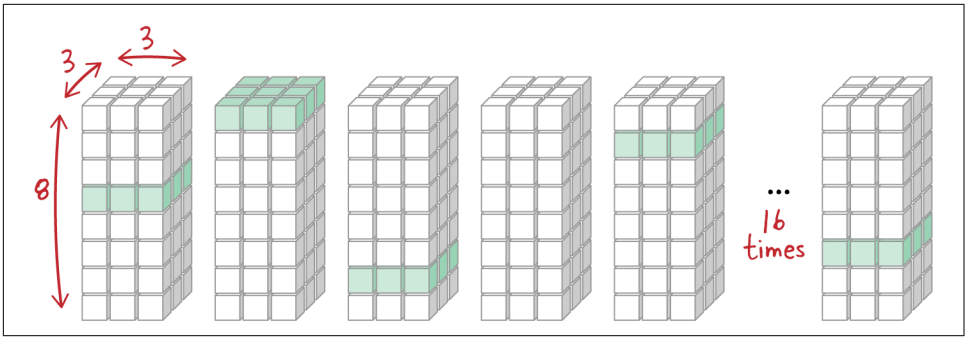


Figure 3-34. The weights of a 3x3 convolutional layer with 8 inputs and 16 outputs (16 filters). Many of the individual 3x3 filters are likely to be similar after training (shaded); for example, a horizontal line detector filter.

In each 3x3x8 filter, there are really two operations happening simultaneously: a 3x3 filter is applied to every input channel across the height and width of the image (spatial dimensions), and filtered outputs are recombined in various ways across channels. In short, the two operations are spatial filtering, combined with a linear recombination of the filtered outputs. If these two operations turned out to be independent (or separable), without affecting the performance of the network, they could be performed with fewer learnable weights. Let's see why.

If we look at the 16 filters of a trained layer, it is probable that the network had to reinvent the same 3x3 spatial filters in many of them, just because it wanted to combine them in different ways. In fact, this can be visualized experimentally (Figure 3-35).

Figure 3-35. Visualization of some of the 12x12 filters from the first convolutional layer of a trained neural network. Very similar filters have been reinvented multiple times. Image from Sifre, 2014.

It looks like traditional convolutional layers use parameters inefficiently. That is why Laurent Sifre suggested, in section 6.2 of his 2014 thesis "Rigid-Motion Scattering for Image Classification," to use a different kind of convolution called a *depth-separable convolution*, or just a *separable convolution*. The main idea is to filter the input, channel by channel, using a set of independent filters, and then combine the outputs separately using 1x1 convolutions, as depicted in Figure 3-36. The hypothesis is that there is little "shape" information to be extracted across channels, and therefore a weighted sum is all that is needed to combine them (a 1x1 convolution is a weighted sum of

channels). On the other hand, there is a lot of "shape" information in the spatial dimensions of the image, and 3x3 filters or bigger are needed to catch it.

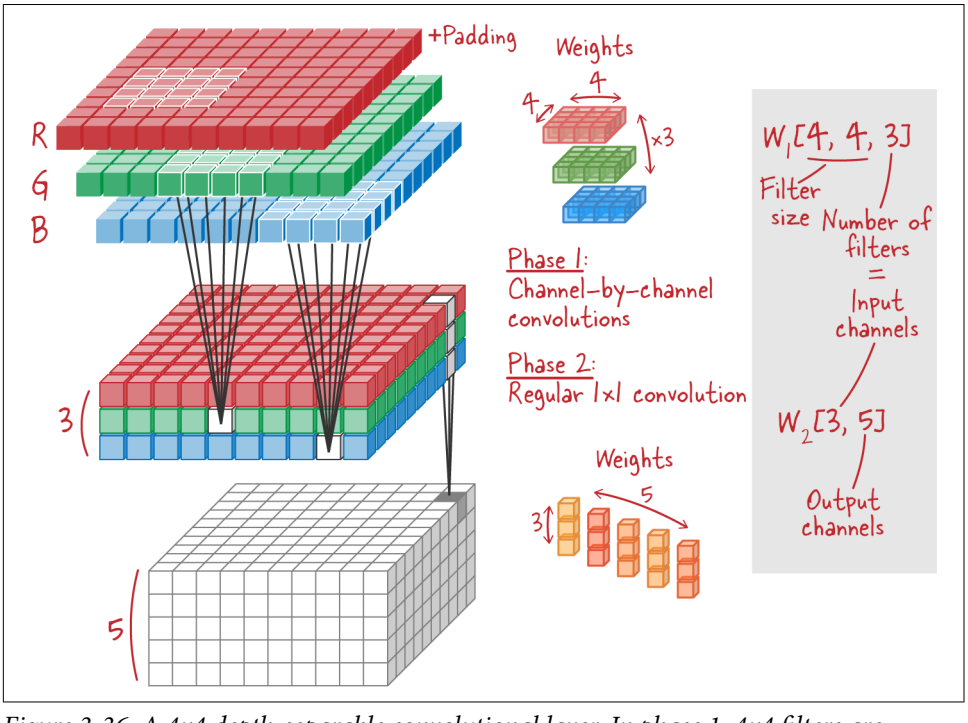


Figure 3-36. A 4x4 depth-separable convolutional layer. In phase 1, 4x4 filters are applied independently to each channel, producing an equal number of output channels. In phase 2, the output channels are then recombined using a 1x1 convolution (multiple weighted sums of the channels).

In Figure 3-36, the phase 1 filtering operation can be repeated with new filter weights to produce double or triple the number of channels. This is called a *depth multiplier*, but its usual value is 1, which is why this parameter was not represented in the calculation of the number of weights on the right.

The number of weights used by the example convolutional layer in Figure 3-36 can easily be computed:

- Using a separable 3x3x8x16 convolutional layer: 3 * 3 * 8 + 8 * 16 = 200 weights
- Using a traditional convolutional layer: 3 * 3 * 8 * 16 = 1,152 weights (for comparison)

Since separable convolutional layers do not need to reinvent each spatial filter multiple times, they are significantly cheaper in terms of learnable weights. The question is whether they are as efficient.

François Chollet argues in his paper "Xception: Deep Learning with Depthwise Separable Convolutions" that separable convolutions are in fact a concept very similar to the Inception modules seen in a previous section. Figure 3-37(A) shows a simplified Inception module with three convolutional paths, each made of a 1x1 convolution followed by a 3x3 convolution. This is exactly equivalent to the representation in Figure 3-37(B), where a single 1x1 convolution outputs three times more channels than previously. Each of those blocks of channels is then picked up by a 3x3 convolution. From there, it only takes a parameter adjustment—namely, increasing the number of 3x3 convolutions—to arrive at Figure 3-37(C), where every channel coming out of the 1x1 convolutions is picked up by its own 3x3 convolution.

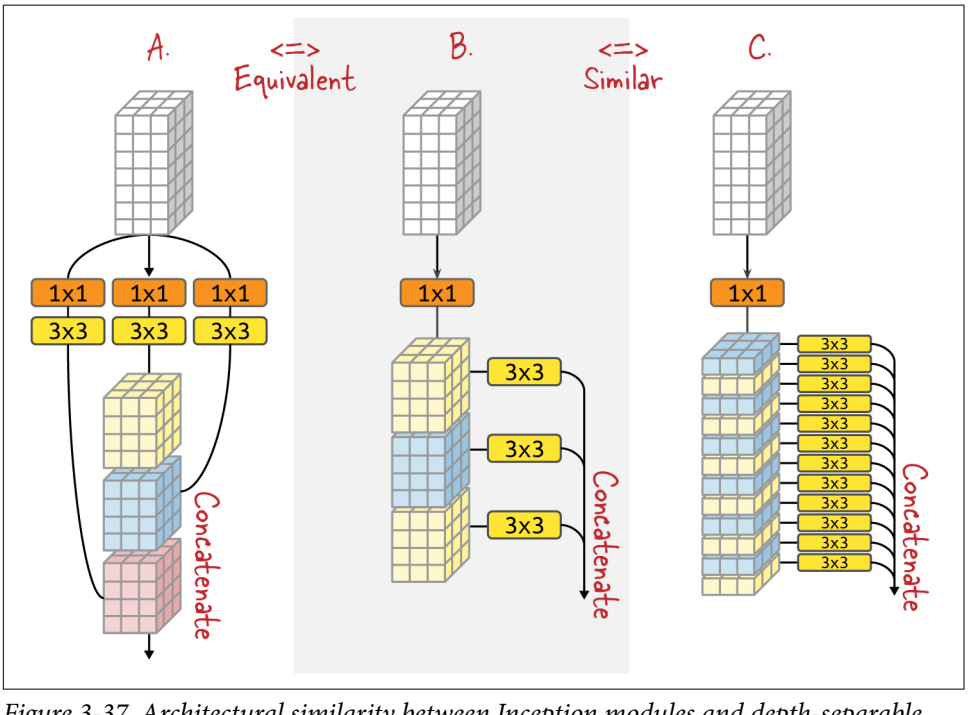


Figure 3-37. Architectural similarity between Inception modules and depth-separable convolutions: (A) a simplified Inception module with three parallel convolutional paths; (B) an exactly equivalent setup where there is a single 1x1 convolution but it outputs three times more channels; (C) a very similar setup with more 3x3 convolutions. This is exactly a separable convolution with the order of the 1x1 and 3x3 operations swapped.

Figure 3-37(C) actually represents a depth-separable convolution with the 1x1 (depthwise) and 3x3 (spatial) operations swapped around. In a convolutional archi-

tecture where these layers are stacked, this change in ordering does not matter much. In conclusion, a simplified Inception module is very similar in its functionality to a depth-separable convolution. This new building block is going to make convolutional architectures both simpler and more economical in terms of learnable weights.

Separable convolutional layers are available in Keras:

```
tf.keras.layers.SeparableConv2D(filters,
                                kernel_size,
                                strides=(1, 1),
                                padding='valid',
                                depth_multiplier=1)
```

The new parameter, compared to a traditional convolutional layer, is the `depth_multiplier` parameter. The following is a simplified description of the parameters (see the Keras documentation for full details):

`filters`
: The number of output channels to produce in the final 1x1 convolution.

`kernel_size`
: The size of each spatial filter. This can be a single number, like 3 for a 3x3 filter, or a pair like (4, 2) for a rectangular 4x2 filter.

`strides`
: The step of the convolution for the spatial filtering.

`padding`
: `'valid'` for no padding, or `'same'` for zero-padding.

`depth_multiplier`
: The number of times the spatial filtering operation is repeated. Defaults to 1.

Xception

The Xception architecture (Figure 3-38) combines separable convolutions with ResNet-style skip connections. Since separable convolutions are somewhat equivalent to Inception-style branching modules, Xception offers a combination of both ResNet and Inception architectural features in a simpler design. Xception's simplicity makes it a good choice when you want to implement your own convolutional backbone. The source code for the Keras implementation of Xception is easily accessible from the documentation.

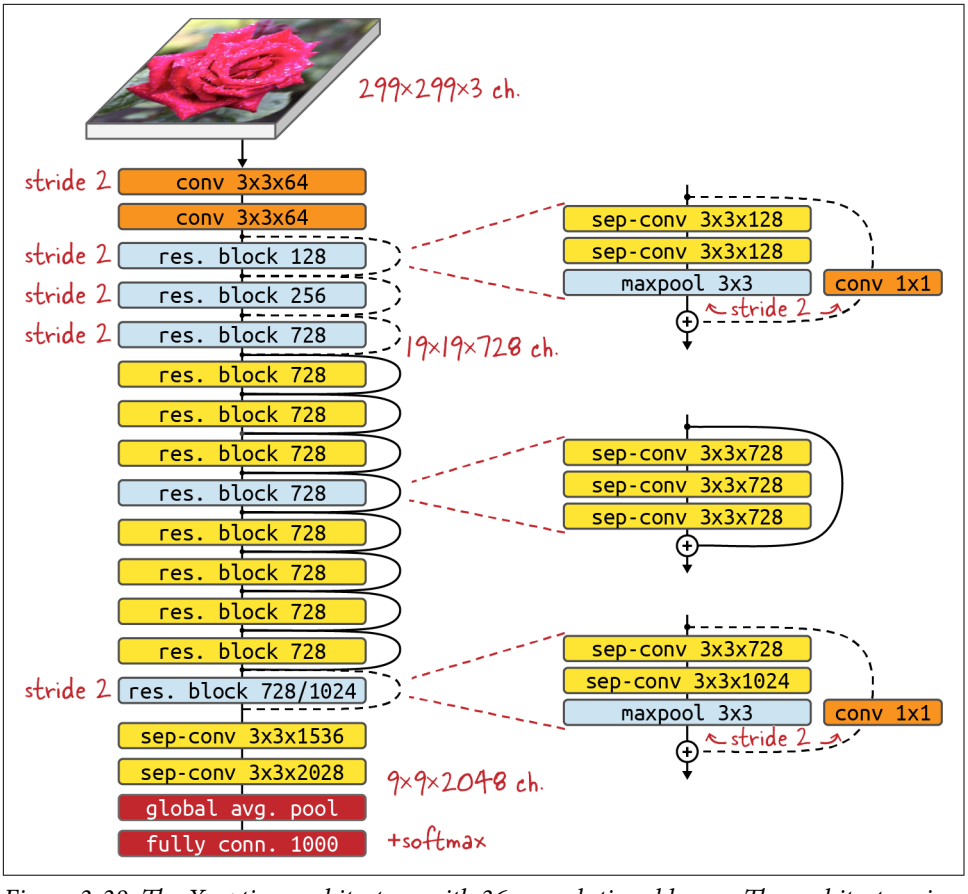


Figure 3-38. The Xception architecture with 36 convolutional layers. The architecture is inspired by ResNet but uses separable convolutions instead of traditional ones, except in the first two layers.

In Figure 3-38, all convolutional layers are ReLU-activated and use batch normalization. All separable convolutions use a depth multiplier of 1 (no channel expansion).

The residual blocks in Xception are different from their ResNet counterparts: they use 3x3 separable convolutions instead of the mix of 3x3 and 1x1 traditional convolutions in ResNet. This makes sense since 3x3 separable convolutions are already a combination of 3x3 and 1x1 convolutions (see Figure 3-36). This further simplifies the design.

It is also to be noted that although depth-separable convolutions have a depth multiplier parameter that allows the initial 3x3 convolutions to be applied multiple times to each input channel with independent weights, the Xception architecture obtains good results with a depth multiplier of 1. This is actually the most common practice. All

other architectures described in this chapter that are based on depth-separable convolutions use them without the depth multiplier (leaving it at 1). It seems that adding parameters in the 1x1 part of the separable convolution is enough to allow the model to capture the relevant information in input images.

Xception at a Glance

Architecture
: Residual blocks based on depth-separable convolutional layers

Publication
: François Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," 2016, *https://arxiv.org/abs/1610.02357*.

Code samples
: *03i_finetune_XCEPTION_flowers104.ipynb* and *03i_fromzero_XCEPTION_flowers104.ipynb*

Table 3-8. Xception at a glance

Model	Parameters (excl. classif. head[a])	ImageNet accuracy	104 flowers F1 score[b] (fine-tuning)	104 flowers F1 score (trained from scratch)
Xception	21M	79%	95% prec.: 95%, recall: 95%	83% prec.: 84%, recall: 82%
Previous best for comparison:				
DenseNet201	18M	77%	95% prec.: 96%, recall: 95%	
DenseNet121	7M	75%		76% prec.: 80%, recall: 74%
InceptionV3	22M	78%	95% prec.: 95%, recall: 94%	
SqueezeNet, 24 layers	2.7M			76% prec.: 77%, recall: 75%

[a] Excluding classification head from parameter counts for easier comparisons between architectures. Without the classification head, the number of parameters in the network is resolution-independent. Also, in fine-tuning examples, a different classification head might be used.
[b] For accuracy, precision, recall, and F1 score values, higher is better.

Neural Architecture Search Designs

The convolutional architectures described in the previous pages are all made of similar elements arranged in different ways: 3x3 and 1x1 convolutions, 3x3 separable convolutions, additions, concatenations… Couldn't the search for the ideal combination be automated? Let's look at architectures that can do this.

NASNet

Automating the search for the optimal combination of operations is precisely what the authors of the NASNet paper did. However, a brute-force search through the entire set of possible operations would have been too large a task. There are too many ways to choose and assemble layers into a full neural network. Furthermore, each piece has many hyperparameters, like the number of its output channels or the size of its filters.

Instead, they simplified the problem in a clever way. Looking back at the Inception, ResNet, or Xception architectures (Figures 3-24, 3-29, and 3-38, respectively), it is easy to see that they are constructed from two types of repeated modules: one kind that leaves the width and height of the features intact ("normal cells") and another that divides them in half ("reduction cells"). The NASNet authors used an automated algorithm to design the structure of these basic cells (see Figure 3-39) and then assembled a convolutional architecture by hand, by stacking the cells with reasonable parameters (channel depth, for example). They then trained the resulting networks to see which module design worked the best.

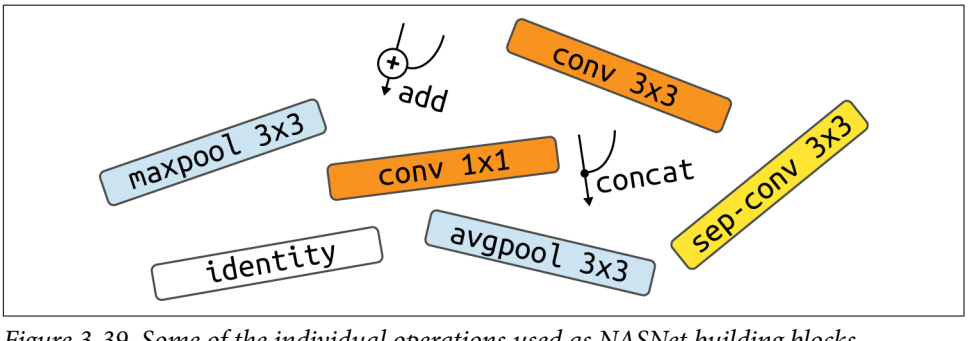


Figure 3-39. Some of the individual operations used as NASNet building blocks.

The search algorithm can either be a random search, which actually did not perform so badly in the study, or a more sophisticated one also based on neural networks called *reinforcement learning*. To learn more about reinforcement learning, see Andrej Karpathy's "Pong from Pixels" post or Martin Görner's and Yu-Han Liu's "Reinforcement Learning Without a PhD" Google I/O 2018 video.

Figure 3-40 shows the structure of the best normal and reduction cells found by the algorithm. Note that the search space allowed connections from not only the previous stage but also the one before, to mimic more densely connected architectures like DenseNet.

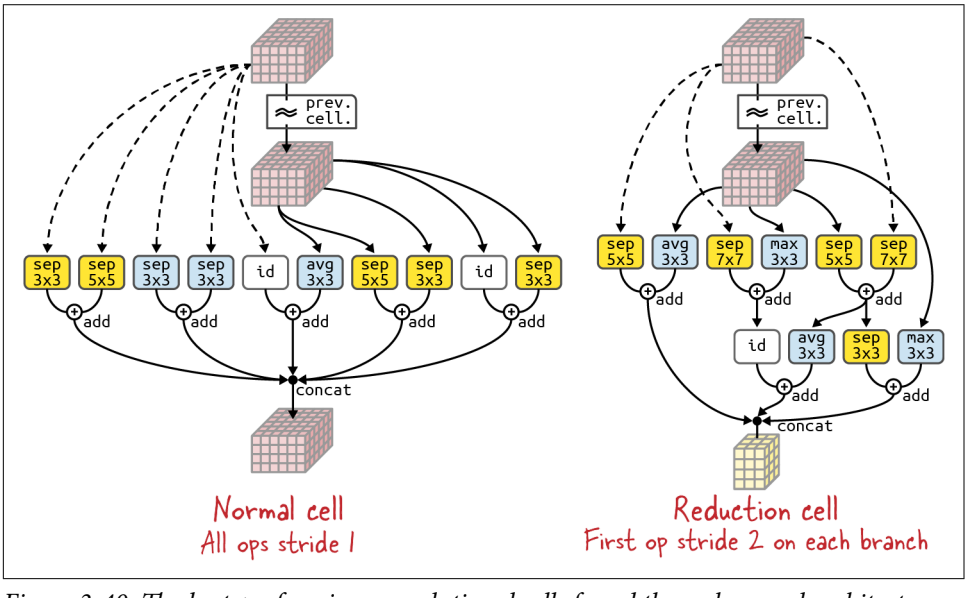


Figure 3-40. The best-performing convolutional cells found through neural architecture search in the NASNet paper. They are made of separable convolutions well as average and max-pooling layers.

The paper notes that separable convolutions are always used doubled ("sep 3x3" in Figure 3-40 actually indicates two consecutive 3x3 separable convolutions), which has been empirically found to increase performance.

Figure 3-41 shows how the cells are stacked to form a complete neural network.

Different NASNet scales can be obtained by adjusting the *N* and *M* parameters—for example, *N*=7 and *M*=1,920 for the most widely used variant, which has 22.6M parameters. All convolutional layers in the figure are ReLU-activated and use batch normalization.

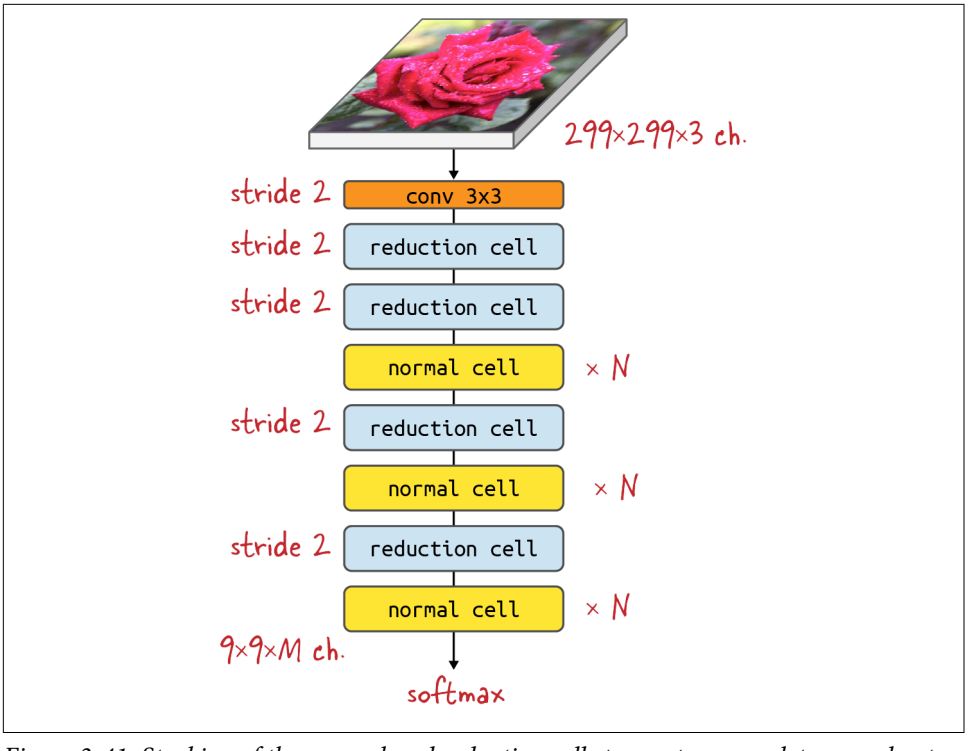


Figure 3-41. Stacking of the normal and reduction cells to create a complete neural network. Normal cells are repeated N times. The number of channels is multiplied by 2 in every reduction cell to obtain M output channels at the end.

There are a few interesting details to note about what the algorithm does:

- It only uses separable convolutions, although regular convolutions were part of the search space. This seems to confirm the benefits of separable convolutions.
- When merging branches, the algorithm chooses to add results rather than concatenate them. This is similar to ResNet but unlike Inception or DenseNet, which use concatenations. (Note that the last concatenation in each cell is forced by the architecture and was not chosen by the algorithm.)
- In the normal cell, the algorithm chooses multiple parallel branches, rather than fewer branches, and more layers of transformations. This is more like Inception, and less like ResNet.
- The algorithm uses separable convolutions with large 5x5 or 7x7 filters rather than implementing everything with 3x3 convolutions. This runs contrary to the "filter factorization" hypothesis outlined earlier in this chapter, and indicates that the hypothesis might not hold after all.

Some choices seem dubious, though, and are probably an artifact of the search space design. For example, in the normal cell, 3x3 average-pooling layers with a stride of 1 are basically blur operations. Maybe a blur is useful, but blurring the same input twice then adding the results is certainly not optimal.

NASNet at a Glance

Architecture
Complex, machine-generated

Publication
Barret Zoph et al., "Learning Transferable Architectures for Scalable Image Recognition," 2017, *https://arxiv.org/abs/1707.07012*.

Code sample
03j_finetune_NASNETLARGE_flowers104.ipynb

Table 3-9. NASNet at a glance

Model	Parameters (excl. classification head[a])	ImageNet accuracy	104 flowers F1 score[b] (fine-tuning)
NASNetLarge	85M	82%	89% precision: 92%, recall: 89%
Previous best for comparison:			
DenseNet201	18M	77%	95% precision: 96%, recall: 95%
Xception	21M	79%	95% precision: 95%, recall: 95%

[a] Excluding classification head from parameter counts for easier comparisons between architectures. Without the classification head, the number of parameters in the network is resolution-independent. Also, in fine-tuning examples, a different classification head might be used.
[b] For accuracy, precision, recall, and F1 score values, higher is better.

Despite its hefty weight count, NASNetLarge is not the best model for the 104 flowers dataset (achieving an F1 score of 89% versus 95% for the other models). This is probably because a large set of trainable parameters requires more training data.

The MobileNet Family

In the next couple of sections, we will describe the MobileNetV2/MnasNet/EfficientNet family of architectures. MobileNetV2 fits in this "neural architecture search" section because it introduces new building blocks that help design a more efficient search space. Although the initial MobileNetV2 was designed by hand, follow-up versions MnasNet and EfficientNet use the same building blocks for automated neural architecture search and end up with optimized but very similar architectures. Before we discuss this set of architectures, however, first we need to introduce two new building blocks: depthwise convolutions and inverted residual bottlenecks.

Depthwise convolutions

The first building block we need to explain in order to understand the MobileNetV2 architecture is depthwise convolutions. The MobileNetV2 architecture revisits depth-separable convolutions and their interactions with skip connections. To make this fine-grained analysis possible, we must first split the depth-separable convolutions described previously (Figure 3-36) into their basic components:

- The spatial filtering part, called a depthwise convolution (Figure 3-42)
- A 1x1 convolution

In Figure 3-42, the filtering operation can be repeated with new filter weights to produce double or triple the number of channels. This is called a "depth multiplier" but its usual value is 1, which is why it is not represented in the picture.

Depthwise convolutional layers are available in Keras:

```
tf.keras.layers.DepthwiseConv2D(kernel_size,
                                strides=(1, 1),
                                padding='valid',
                                depth_multiplier=1)
```

Note that a depth-separable convolution, such as:

```
tf.keras.layers.SeparableConv2D(filters=128, kernel_size=(3,3))
```

can also be represented in Keras as a sequence of two layers:

```
tf.keras.layers.DepthwiseConv2D(kernel_size=(3,3)
tf.keras.layers.Conv2D(filters=128, kernel_size=(1,1))
```

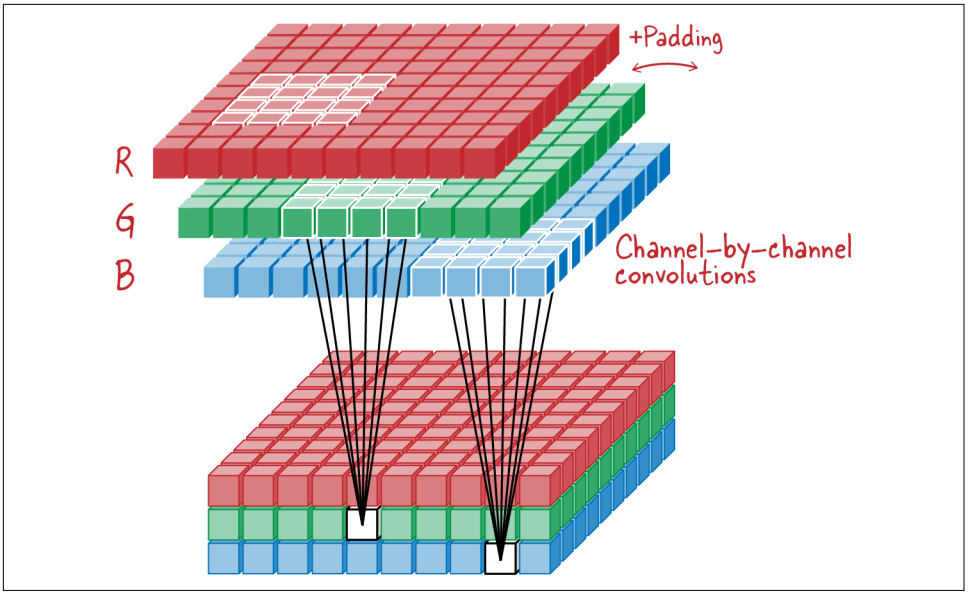


Figure 3-42. A depthwise convolutional layer. Convolutional filters are applied independently to each input channel, producing an equal number of output channels.

Inverted residual bottlenecks

The second and most important building block in the MobileNet family is the inverted residual bottleneck. Residual blocks used in ResNet or Xception architectures tend to keep the number of channels flowing through skip connections high (see Figure 3-43 below). In the MobileNetV2 paper, the authors make the hypothesis that the information that skip connections help preserve is inherently low-dimensional. This makes sense intuitively. If a convolutional block specializes in detecting, for example "cat whiskers," the information in its output ("whiskers detected at position (3, 16)") can be represented along three dimensions: class, *x*, *y*. Compared with the pixel representation of whiskers, it is low-dimensional.

The MobileNetV2 architecture introduces a new residual block design that places skip connections where the number of channels is low and expands the number of channels inside residual blocks. Figure 3-43 compares the new design to the typical residual blocks used in ResNet and Xception. The number of channels in ResNet blocks follows the sequence "many – few – many," with skip connections between the "many channels" stages. Xception does "many – many – many." The new MobileNetV2 design follows the sequence "few – many – few." The paper calls this technique *inverted residual bottlenecks*—"inverted" because it is the exact opposite of the ResNet approach, "bottleneck" because the number of channels is squeezed in between residual blocks, like the neck of a bottle.

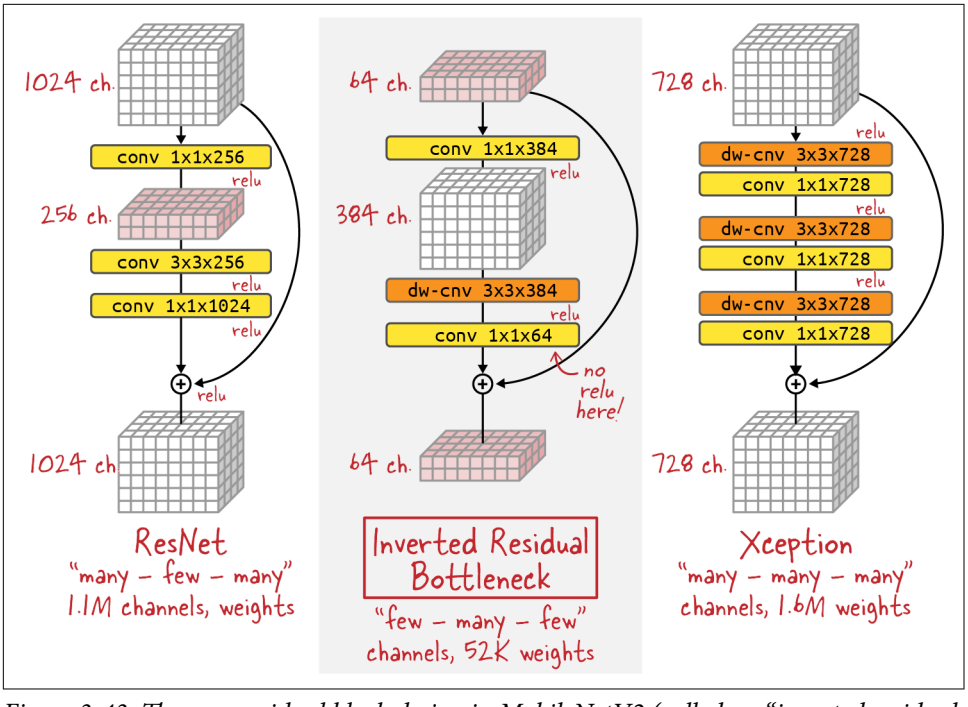


Figure 3-43. The new residual block design in MobileNetV2 (called an "inverted residual bottleneck"), compared with ResNet and Xception residual blocks. "dw-cnv" stands for depthwise convolution. Separable convolutions used by Xception are represented by their components: "dw-cnv" followed by "conv 1x1."

The goal of this new residual block is to offer the same expressivity as prior designs with a dramatically reduced weight count and, even more importantly, a reduced latency at inference time. MobileNetV2 was indeed designed to be used on mobile phones, where compute resources are scarce. The weight counts of the typical residual blocks represented in Figure 3-43 are 1.1M, 52K, and 1.6M respectively for the ResNet, MobileNetV2, and Xception blocks.

The authors of the MobileNetV2 paper argue that their design can achieve good results with fewer parameters because the information that flows between residual blocks is low-dimensional in nature and can therefore be represented in a limited number of channels. However, one construction detail is important: the last 1x1 convolution in the inverted residual block, the one that squeezes the feature map back to "few" channels, is not followed by a nonlinear activation. The MobileNetV2 paper covers this topic at some length, but the short version is that in a low-dimensional space, a ReLU activation would destroy too much information.

We are now ready to build a full MobileNetV2 model and then use neural architecture search to refine it into the optimized, but otherwise very similar, MnasNet and EfficientNet architectures.

MobileNetV2

We can now put together the MobileNetV2 convolutional stack. MobileNetV2 is built out of multiple inverted residual blocks, as shown in Figure 3-44.

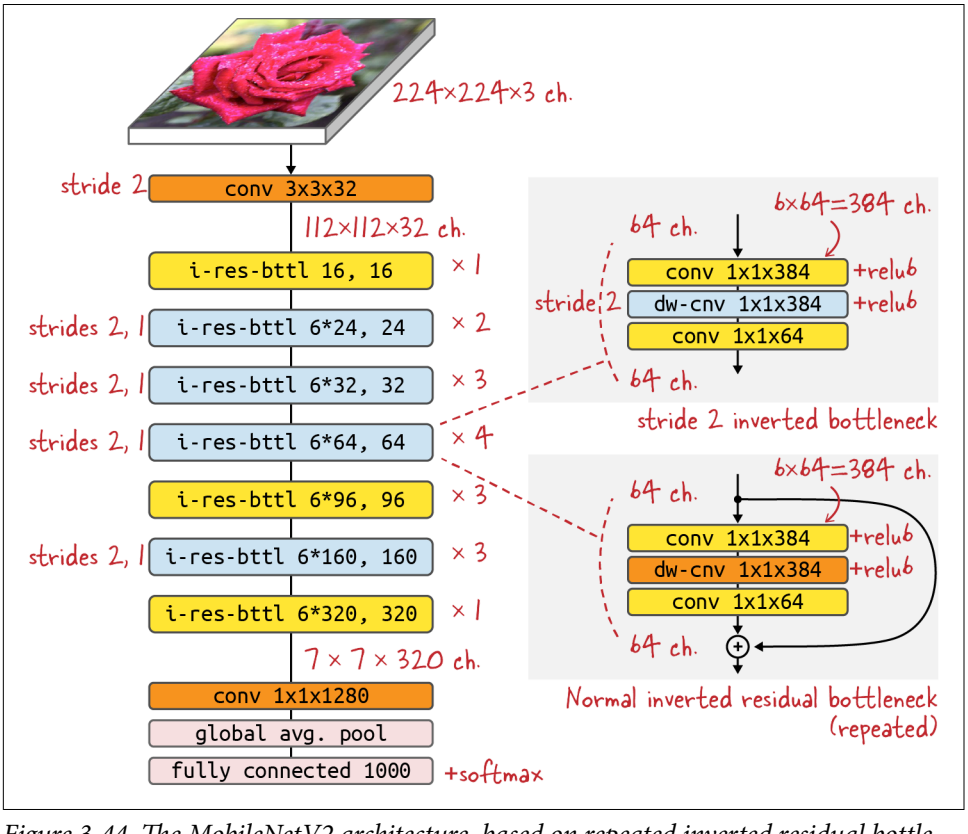


Figure 3-44. The MobileNetV2 architecture, based on repeated inverted residual bottlenecks. Repeat counts are in the center column. "conv" indicates regular convolutional layers, while "dw-cnv" denotes depthwise convolutions.

In Figure 3-44, inverted residual bottleneck blocks are marked "i-res-bttl *N*, *M*" and parameterized by their internal (*N*) and external channel depth (*M*). Every sequence marked "strides 2, 1" starts with an inverted bottleneck block with a stride of 2 and no skip connection. The sequence continues with regular inverted residual bottleneck blocks. All convolutional layers use batch normalization. Please note that the last convolutional layer in inverted bottleneck blocks does not use an activation function.

The activation function in MobileNetV2 is ReLU6, instead of the usual ReLU. Later evolutions of MobileNetV2 went back to using standard ReLU activation functions. The use of ReLU6 in MobileNetV2 is not a fundamental implementation detail.

MobileNetV2 at a Glance

Architecture

Sequence of inverted residual bottlenecks

Publication

Mark Sandler et al., "MobileNetV2: Inverted Residuals and Linear Bottlenecks," 2018, *https://arxiv.org/abs/1801.04381*.

Code sample

03k_finetune_MOBILENETV2_flowers104.ipynb

Table 3-10. MobileNetV2 at a glance

Model	Parameters (excl. classification head[a])	ImageNet accuracy	104 flowers F1 score[b] (fine-tuning)
MobileNetV2	2.3M	71%	92% precision: 92%, recall: 92%
Previous best for comparison:			
NASNetLarge	85M	82%	89% precision: 92%, recall: 89%
DenseNet201	18M	77%	95% precision: 96%, recall: 95%
Xception	21M	79%	95% precision: 95%, recall: 95%

[a] Excluding classification head from parameter counts for easier comparisons between architectures. Without the classification head, the number of parameters in the network is resolution-independent. Also, in fine-tuning examples, a different classification head might be used.

[b] For accuracy, precision, recall, and F1 score values, higher is better.

MobileNetV2 is optimized for a low weight count and sacrifices a bit of accuracy for it. Also, in the 104 flowers fine-tuning example, it converged significantly slower than other models. It can still be a good choice when mobile inference performance is important.

The MobileNetV2 simple structure of repeated inverted residual bottleneck blocks lends itself well to automated neural architecture search methods. That is how the MnasNet and EfficientNet architectures were created.

EfficientNet: Putting it all together

The team that created MobileNetV2 later refined the architecture through automated neural architecture search, using inverted residual bottlenecks as the building blocks of their search space. The MnasNet paper summarizes their initial findings. The most interesting result of that research is that, once again, the automated algorithm reintroduced 5x5 convolutions into the mix. This was already the case in NASNet, as we saw earlier. This is interesting because all manually constructed architectures had standardized on 3x3 convolutions, justifying the choice with the filter factorization hypothesis. Apparently, larger filters like 5x5 are useful after all.

We'll skip a formal description of the MnasNet architecture in favor of its next iteration: EfficientNet. This architecture was developed using the exact same search space and network architecture search algorithm as MnasNet, but the optimization goal was tweaked toward prediction accuracy rather than mobile inference latency. Inverted residual bottlenecks from MobileNetV2 are again the basic building blocks.

EfficientNet is actually a family of neural networks of different sizes, where a lot of attention was paid to the scaling of the networks in the family. Convolutional architectures have three main ways of scaling:

- Use more layers.
- Use more channels in each layer.
- Use higher-resolution input images.

The EfficientNet paper points out that these three scaling axes are not independent: "If the input image is bigger, then the network needs more layers to increase the receptive field and more channels to capture more fine-grained patterns on the bigger image."

The novelty in the EfficientNetB0 through EfficientNetB7 family of neural networks is that they are scaled along all three scaling axes rather than just one, as was the case in earlier architecture families such as ResNet50/ResNet101/ResNet152. The EfficientNet family is today the workhorse of many applied machine learning teams because it offers optimal performance levels for every weight count. Research evolves fast though, and by the time this book is printed, it is probable that an even better architecture will have been discovered.

Figure 3-45 describes the baseline EfficientNetB0 architecture. Notice the similarity with MobileNetV2.

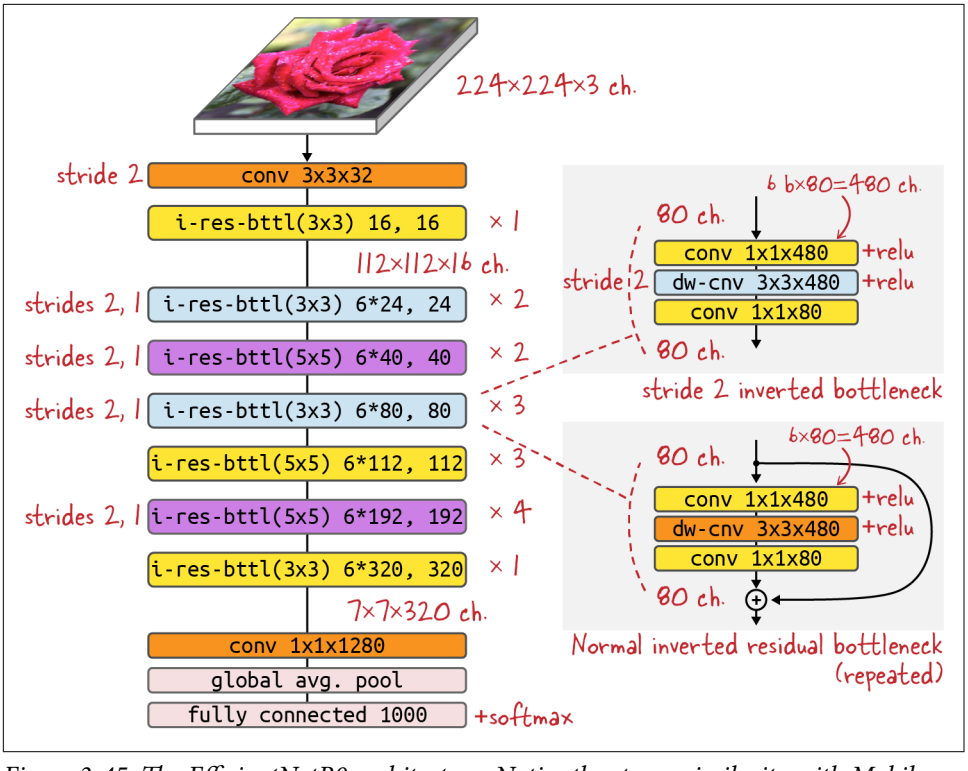


Figure 3-45. The EfficientNetB0 architecture. Notice the strong similarity with MobileNetV2 (Figure 3-44).

In Figure 3-45, sequences of inverted residual bottlenecks are noted [i-res-bttl(*K*x*K*) *P***Ch*, *Ch*] × *N*, where:

- *Ch* is the external number of channels output by each block.
- The internal number of channels is typically a multiple *P* of the external channels: *P***Ch*.
- *K*x*K* is the convolutional filter size, typically 3x3 or 5x5.
- *N* is the number of such consecutive layer blocks.

Every sequence marked "strides 2, 1" starts with an inverted bottleneck block with a stride of 2 and no skip connection. The sequence continues with regular inverted residual bottleneck blocks. As previously mentioned, "conv" indicates regular convolutional layers, while "dw-cnv" denotes depthwise convolutions.

EfficientNetB1 through B7 have the exact same general structure, with seven sequences of inverted residual bottlenecks; only the parameters differ. Figure 3-46 provides the scaling parameters for the entire family.

EfficientNetB0	EfficientNetB1	EfficientNetB2
ideal image res.: 224x224 px	ideal image res.: 240x240 px	ideal image res.: 260x260 px
weight count: 5.3M	weight count: 7.9M	weight count: 9.2M
[i-res-bttl(3x3) 16, 16] x 1	[i-res-bttl(3x3) 16, 16] x 2	[i-res-bttl(3x3) 16, 16] x 2
[i-res-bttl(3x3) 144, 24] x 2	[i-res-bttl(3x3) 144, 24] x 3	[i-res-bttl(3x3) 144, 24] x 3
[i-res-bttl(5x5) 240, 40] x 2	[i-res-bttl(5x5) 240, 40] x 3	[i-res-bttl(5x5) 288, 48] x 3
[i-res-bttl(3x3) 480, 80] x 3	[i-res-bttl(3x3) 480, 80] x 4	[i-res-bttl(3x3) 528, 88] x 4
[i-res-bttl(5x5) 672, 112] x 3	[i-res-bttl(5x5) 672, 112] x 4	[i-res-bttl(5x5) 720, 120] x 4
[i-res-bttl(5x5) 1152, 192] x 4	[i-res-bttl(5x5) 1152, 192] x 5	[i-res-bttl(5x5) 1248, 208] x 5
[i-res-bttl(3x3) 1920, 320] x 1	[i-res-bttl(3x3) 1920, 320] x 2	[i-res-bttl(3x3) 2112, 352] x 2

EfficientNetB3	EfficientNetB4	EfficientNetB5
ideal image res.: 300x300 px	ideal image res.: 380x380 px	ideal image res.: 456x456 px
weight count: 12.3M	weight count: 19.5M	weight count: 30.6M
[i-res-bttl(3x3) 24, 24] x 2	[i-res-bttl(3x3) 24, 24] x 2	[i-res-bttl(3x3) 24, 24] x 3
[i-res-bttl(3x3) 192, 32] x 3	[i-res-bttl(3x3) 192, 32] x 4	[i-res-bttl(3x3) 240, 40] x 5
[i-res-bttl(5x5) 288, 48] x 3	[i-res-bttl(5x5) 336, 56] x 4	[i-res-bttl(5x5) 384, 64] x 5
[i-res-bttl(3x3) 576, 96] x 5	[i-res-bttl(3x3) 672, 112] x 6	[i-res-bttl(3x3) 768, 128] x 7
[i-res-bttl(5x5) 816, 136] x 5	[i-res-bttl(5x5) 960, 160] x 6	[i-res-bttl(5x5) 1056, 176] x 7
[i-res-bttl(5x5) 1392, 232] x 6	[i-res-bttl(5x5) 1632, 272] x 8	[i-res-bttl(5x5) 1824, 304] x 9
[i-res-bttl(3x3) 2304, 384] x 2	[i-res-bttl(3x3) 2688, 448] x 2	[i-res-bttl(3x3) 3072, 512] x 3

EfficientNetB6	EfficientNetB7	EfficientNetB8
ideal image res.: 528x528 px	ideal image res.: 600x600 px	ideal image res.: 672x672 px
weight count: 43.3M	weight count: 66.7M	
[i-res-bttl(3x3) 32, 32] x 3	[i-res-bttl(3x3) 32, 32] x 4	[i-res-bttl(3x3) 32, 32] x 4
[i-res-bttl(3x3) 240, 40] x 6	[i-res-bttl(3x3) 288, 48] x 7	[i-res-bttl(3x3) 336, 56] x 8
[i-res-bttl(5x5) 432, 72] x 6	[i-res-bttl(5x5) 480, 80] x 7	[i-res-bttl(5x5) 528, 88] x 8
[i-res-bttl(3x3) 864, 144] x 8	[i-res-bttl(3x3) 960, 160] x 10	[i-res-bttl(3x3) 1056, 176] x 11
[i-res-bttl(5x5) 1200, 200] x 8	[i-res-bttl(5x5) 1344, 224] x 10	[i-res-bttl(5x5) 1488, 248] x 11
[i-res-bttl(5x5) 2064, 344] x 11	[i-res-bttl(5x5) 2304, 384] x 13	[i-res-bttl(5x5) 2544, 424] x 15
[i-res-bttl(3x3) 3456, 576] x 3	[i-res-bttl(3x3) 3840, 640] x 4	[i-res-bttl(3x3) 4224, 704] x 4

Figure 3-46. The EfficientNetB0 through EfficientNetB7 family, showing the parameters of the seven sequences of inverted residual bottlenecks that make up the EfficientNet architecture.

As shown in Figure 3-46, each neural network in the family has an ideal input image size. It has been trained on images of this size, though it can also be used with other image sizes. The number of layers and number of channels in each layer are scaled along with the input image size. The multiplier between the external and internal number of channels in inverted residual bottlenecks is always 6, apart from in the first row where it is 1.

So are these scaling parameters actually effective? The EfficientNet paper shows they are. The compound scaling outlined above is more efficient than scaling the network by layers, channels, or image resolution alone (Figure 3-47).

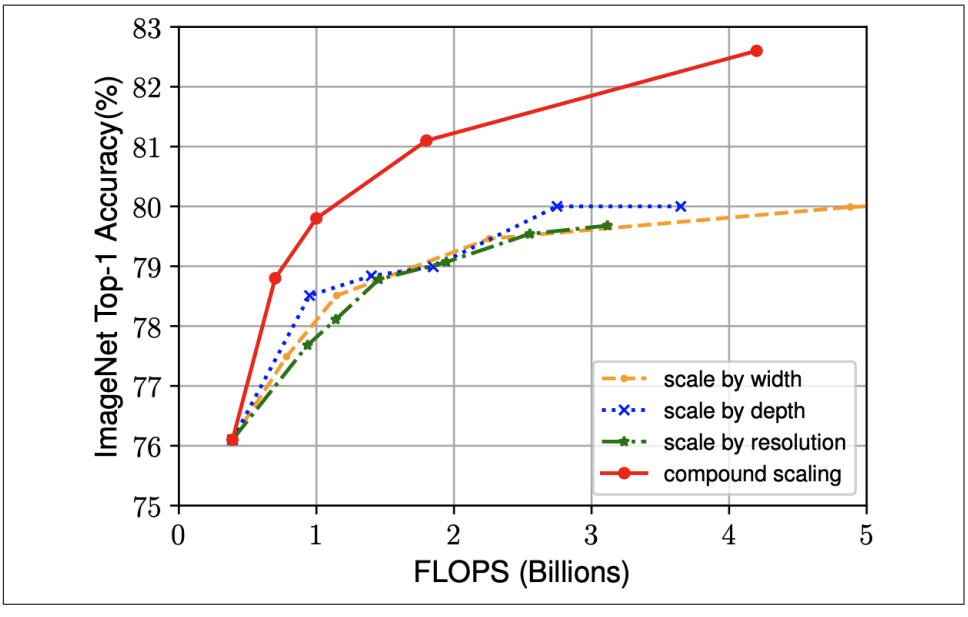


Figure 3-47. Accuracy of EfficientNet classifiers scaled using the compound scaling method from the EfficientNet paper versus scaling by a single factor: width (the number of channels in convolutional blocks), depth (the number of convolutional layers), or image resolution. Image from Tan & Le, 2019.

The authors of the EfficientNet paper also used the class activation map technique from Zhou et al., 2016 to visualize what the trained networks "see." Again, compound scaling achieves better results by helping the network focus on the important parts of the image (Figure 3-48).

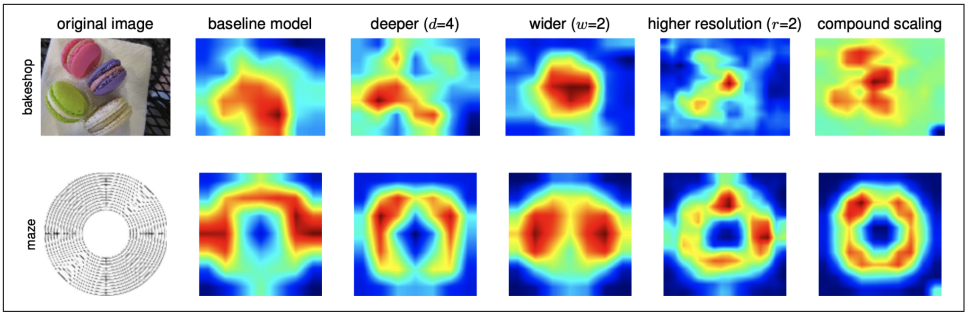


Figure 3-48. Class activation maps (Zhou et al., 2016) for two input images as seen through several EfficientNet variants. The model obtained through compound scaling (last column) focuses on more relevant regions with more object detail. Image from Tan & Le, 2019.

EfficientNet also incorporates some additional optimizations. Briefly:

- Every inverted bottleneck block is further optimized through the "squeeze-excite" channel optimization, as per Jie et al., 2017. This technique is a channel-wise attention mechanism that "renormalizes" output channels (i.e., boosts some and attenuates others) before the final 1x1 convolution of each block. Like any "attention" technique, it involves a small additional neural network that learns to produce the ideal renormalization weights. This additional network is not represented in Figure 3-45. Its contribution to the total count of learnable weights is small. This technique can be applied to any convolutional block, not just inverted residual bottlenecks, and boosts network accuracy by about one percentage point.
- Dropout is used in all members of the EfficientNet family to help with overfitting. Larger networks in the family use slightly larger dropout rates (0.2, 0.2, 0.3, 0.3, 0.4, 0.4, 0.5, and 0.5, respectively, for EfficientNetB0 through B7).
- The activation function used in EfficientNet is SiLU (also called Swish-1) as described in Ramachandran et al., 2017. The function is $f(x) = x \cdot \text{sigmoid}(x)$.
- The training dataset was automatically expanded using the AutoAugment technique, as described in Cubuk et al., 2018.
- The "stochastic depth" technique is used during training, as described in Huang et al., 2016. We are not sure how effective this part was since the stochastic depth paper itself reports that the technique does not work with a ResNet152 trained on ImageNet. It might do something on deeper networks.

EfficientNet at a Glance

Architecture
: Sequence of inverted residual bottlenecks

Code samples
: *03l_finetune_EFFICIENTNETB6_flowers104.ipynb*, *03l_finetune_EFFICIENTNETB7_TFHUB_flowers104.ipynb*, and *03l_fromzero_EFFICIENTNETB4_ flowers104.ipynb*

Publication
: Mingxing Tan and Quoc V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," 2019, *arXiv:1905.11946*.

EfficientNetB6 and B7 currently top the ImageNet classification charts with an accuracy of 84%. Fine-tuned on our 104 flowers dataset, however, they perform only marginally better than Xception, DenseNet201, or InceptionV3. All of these models tend to achieve precision and recall values of 95% on this dataset and saturate there. The dataset is probably too small to go further.

You will find a table summarizing all the results in the next section.

Beyond Convolution: The Transformer Architecture

The architectures for computer vision that are discussed in this chapter all rely on convolutional filters. Compared to the naive dense neural networks discussed in Chapter 2, convolutional filters reduce the number of weights necessary to learn how to extract information from images. However, as dataset sizes keep increasing, there comes a point where this weight reduction is no longer necessary.

Ashish Vaswani et al. proposed the Transformer architecture for natural language processing in a 2017 paper with the catchy title "Attention Is All You Need." As the title indicates, the key innovation in the Transformer architecture is the concept of *attention*—having the model focus on some part of the input text sequence when predicting each word. For example, consider a model that needs to translate the French phrase "ma chemise rouge" into English ("my red shirt"). The model would learn to focus on the word *rouge* when predicting the second word of the English translation, *red*. The Transformer model achieves this by using *positional encodings*. Instead of simply representing the input phrase by its words, it adds the position of each word as an input: (ma, 1), (chemise, 2), (rouge, 3). The model then learns from the training dataset which word of the input it needs to focus on when predicting a specific word of the output.

The Vision Transformer (ViT) model adapts the Transformer idea to work on images. The equivalent of words in images are square patches, so the first step is to take the

input image and break it into patches, as shown in Figure 3-49 (the full code is available in *03m_transformer_flowers104.ipynb* on GitHub):

```
patches = tf.image.extract_patches(
    images=images,
    sizes=[1, self.patch_size, self.patch_size, 1],
    strides=[1, self.patch_size, self.patch_size, 1],
    rates=[1, 1, 1, 1],
    padding="VALID",
)
```

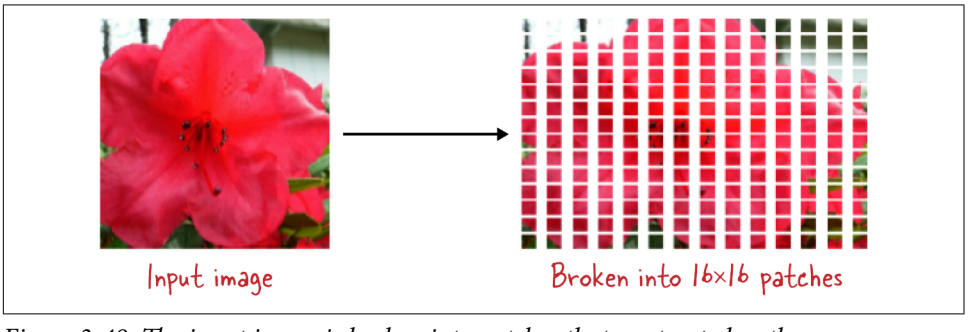


Figure 3-49. The input image is broken into patches that are treated as the sequence input to the Transformer.

The patches are represented by concatenating the patch pixel values and the patch position within the image:

```
encoded = (tf.keras.layers.Dense(...)(patch) +
           tf.keras.layers.Embedding(...)(position))
```

Note that the patch position is the ordinal number (5th, 6th, etc.) of the patch and is treated as a categorical variable. A learnable embedding is employed to capture closeness relationships between patches that have related content.

The patch representation is passed through multiple transformer blocks, each of which consists of an attention head (to learn which parts of the input to focus on):

```
x1 = tf.keras.layers.LayerNormalization()(encoded)
attention_output = tf.keras.layers.MultiHeadAttention(
    num_heads=num_heads, key_dim=projection_dim, dropout=0.1
)(x1, x1)
```

The attention output is used to add emphasis to the patch representation:

```
# Skip connection 1.
x2 = tf.keras.layers.Add()([attention_output, encoded])
# Layer normalization 2.
x3 = tf.keras.layers.LayerNormalization()(x2)
```

and passed through a set of dense layers:

```
    # multilayer perceptron (mlp), a set of dense layers.
    x3 = mlp(x3, hidden_units=transformer_units,
             dropout_rate=0.1)
    # Skip connection 2 forms input to next block
    encoded = tf.keras.layers.Add()([x3, x2])
```

The training loop is similar to that of any of the convolutional network architectures discussed in this chapter. Note that the ViT architecture requires a lot more data than convolutional network models—the authors suggest pretraining the ViT model on large amounts of data and then fine-tuning on smaller datasets. Indeed, training from scratch on the 104 flowers dataset yields only a 34% accuracy.

Even though not particularly promising at present for our relatively small dataset, the idea of applying the Transformer architecture to images is interesting, and a potential source of new innovations in computer vision.

Choosing a Model

This section will provide some tips on choosing a model architecture for your task. First of all, create a benchmark using code-free services to train ML models so that you have a good idea of what kind of accuracy is achievable on your problem. If you are training on Google Cloud, consider Google Cloud AutoML, which utilizes neural architecture search (NAS). If you are using Microsoft Azure, consider Custom Vision AI. DataRobot and H2O.ai employ transfer learning for code-free image classification. It is unlikely that you will get an accuracy that is significantly higher than what these services provide out of the box, so you can use them as a way to quickly do a proof of concept before you invest too much time on an infeasible problem.

Performance Comparison

Let's summarize the performance numbers seen so far, first for fine-tuning (Table 3-11). Notice the new entrant at the bottom, called "ensemble." We will cover this in the next section.

Table 3-11. Eight model architectures fine-tuned on the 104 flowers dataset

Model	Parameters (excl. classification head[a])	ImageNet accuracy	104 flowers F1 score[b] (fine-tuning)
EfficientNetB6	40M	84%	95.5%
EfficientNetB7	64M	84%	95.5%
DenseNet201	18M	77%	95.4%
Xception	21M	79%	94.6%
InceptionV3	22M	78%	94.6%
ResNet50	23M	75%	94.1%
MobileNetV2	2.3M	71%	92%

Model	Parameters (excl. classification head[a])	ImageNet accuracy	104 flowers F1 score[b] (fine-tuning)
NASNetLarge	85M	82%	89%
VGG19	20M	71%	88%
Ensemble	79M (DenseNet210 + Xception + EfficientNetB6)	-	96.2%

[a] Excluding classification head from parameter counts for easier comparisons between architectures. Without the classification head, the number of parameters in the network is resolution-independent. Also, in fine-tuning examples, a different classification head might be used.

[b] For accuracy, precision, recall, and F1 score values, higher is better.

And now for training from scratch (Table 3-12). Since fine-tuning worked much better on the 104 flowers dataset, not all the models have been trained from scratch.

Table 3-12. Six model architectures trained from scratch on the 104 flowers dataset

Model	Parameters (excl. classification head[a])	ImageNet accuracy	104 flowers F1 score[b] (trained from scratch)
Xception	21M	79%	82.6%
SqueezeNet, 24 layers	2.7M	-	76.2%
DenseNet121	7M	75%	76.1%
ResNet50	23M	75%	73%
EfficientNetB4	18M	83%	69%
AlexNet	3.7M	60%	39%

[a] Excluding classification head from parameter counts for easier comparisons between architectures. Without the classification head, the number of parameters in the network is resolution-independent. Also, in fine-tuning examples, a different classification head might be used.

[b] For accuracy, precision, recall, and F1 score values, higher is better.

Xception takes the first spot here, which is a bit surprising since it is not the most recent architecture. Xception's author also noticed in his paper that his model seemed to work better than others when applied to real-world datasets other than ImageNet and other standard datasets used in academia. The fact that the second spot is taken by a SqueezeNet-like model quickly thrown together by the author of the book is significant. When you want to try your own architecture, SqueezeNet is both very simple to code and quite efficient. This model is also the smallest one in the selection. Its size is probably well adapted to the relatively small size of the 104 flowers dataset (approximately 20K pictures). The DenseNet architecture shares the second place with SqueezeNet. It is by far the most unconventional architecture in this selection, but it seems to have a lot of potential on unconventional datasets.

It might be worth looking at the other variations and versions of these models to pick the most suitable and most up-to-date one. As mentioned, EfficientNet was the state-of-the-art model at the time we wrote this book (January 2021). There might be

something newer by the time you are reading it. You can check TensorFlow Hub for new models.

A last option is to use multiple models at the same time, a technique called *ensembling*. We'll look at this next.

Ensembling

When looking for the maximum accuracy, and when model size and inference times are not an issue, multiple models can be used at the same time and their predictions combined. Such *ensemble models* can often give better predictions than any of the models composing them. Their predictions are also more robust on real-life images. The key consideration when selecting models to ensemble is to choose models that are as different as possible from each other. Models with very different architectures are more likely to have different weaknesses. When combined in an ensemble, the strengths and weaknesses of different models will compensate for each other, as long as they are not in the same classes.

A notebook, *03z_ensemble_finetune_flowers104.ipynb*, is provided in the GitHub repository showcasing an ensemble of three models fine-tuned on the 104 flowers dataset: DenseNet210, Xception, and EfficientNetB6. As seen in Table 3-13, the ensemble wins by a respectable margin.

Table 3-13. Comparison of model ensembling versus individual models

Model	Parameters (excl. classification head[a])	ImageNet accuracy	104 flowers F1 score[b] (fine-tuning)
EfficientNetB6	40M	84%	95.5%
DenseNet201	18M	77%	95.4%
Xception	21M	79%	94.6%
Ensemble	79M		96.2%

[a] Excluding classification head from parameter counts for easier comparisons between architectures. Without the classification head, the number of parameters in the network is resolution-independent. Also, in fine-tuning examples, a different classification head might be used.

[b] For accuracy, precision, recall, and F1 score values, higher is better.

The easiest way to ensemble the three models is to average the class probabilities they predict. Another possibility, theoretically better, is to average their logits (the outputs of the last layer before softmax activation) and apply softmax on the averages to compute class probabilities. The sample notebook shows both options. On the 104 flowers dataset, they perform equally.

One point of caution when averaging logits is that logits, contrary to probabilities, are not normalized. They can have very different values in different models. Computing a weighted average instead of a simple average might help in that case. The training dataset should be used to compute the best weights.

Recommended Strategy

Here is our recommended strategy to tackle computer vision problems.

First, choose your training method based on the size of your dataset:

- If you have a very small dataset (less than one thousand images per label), use transfer learning.
- If you have a moderate-sized dataset (one to five thousand images per label), use fine-tuning.
- If you have a large dataset (more than five thousand images per label), train from scratch.

These numbers are rules of thumb and vary depending on the difficulty of the use case, the complexity of your model, and the quality of the data. You may have to experiment with a couple of the options. For example, the 104 flowers dataset has between one hundred and three thousand images per class, depending on the class; fine-tuning was still very effective on it.

Whether you are doing transfer learning, fine-tuning, or training from scratch, you will need to select a model architecture. Which one should you pick?

- If you want to roll your own layers, start with SqueezeNet. It's the simplest model that will perform well.
- For edge devices, you typically want to optimize for models that can be downloaded fast, occupy very little space on the device, and don't incur high latencies during prediction. For a small model that runs fast on low-power devices, consider MobileNetV2.
- If you don't have size/speed restrictions (such as if inference will be done on autoscaling cloud systems) and want the best/fanciest model, consider EfficientNet.
- If you belong to a conservative organization that wants to stick with something tried and true, choose ResNet50 or one of its larger variants.

If training cost and prediction latency are not of concern, or if small improvements in model accuracy bring outside rewards, consider an ensemble of three complementary models.

Summary

This chapter focused on image classification techniques. It first explained how to use pretrained models and adapt them to a new dataset. This is by far the most popular technique and will work if the pretraining dataset and the target dataset share at least some similarities. We explored two variants of this technique: transfer learning, where the pretrained model is frozen and used as a static image encoder; and fine-tuning, where the weights of the pretrained model are used as initial values in a new training run on the new dataset. We then examined the historical and current state-of-the-art image classification architectures, from AlexNet to EfficientNets. All the building blocks of these architectures were explained, starting of course with convolutional layers, to give you a complete understanding of how these models work.

In Chapter 4, we will look at using any of these image model architectures to solve common computer vision problems.

CHAPTER 4

Object Detection and Image Segmentation

So far in this book, we have looked at a variety of machine learning architectures but used them to solve only one type of problem—that of classifying (or regressing) an entire image. In this chapter, we discuss three new vision problems: object detection, instance segmentation, and whole-scene semantic segmentation (Figure 4-1). Other more advanced vision problems like image generation, counting, pose estimation, and generative models are covered in Chapters 11 and 12.

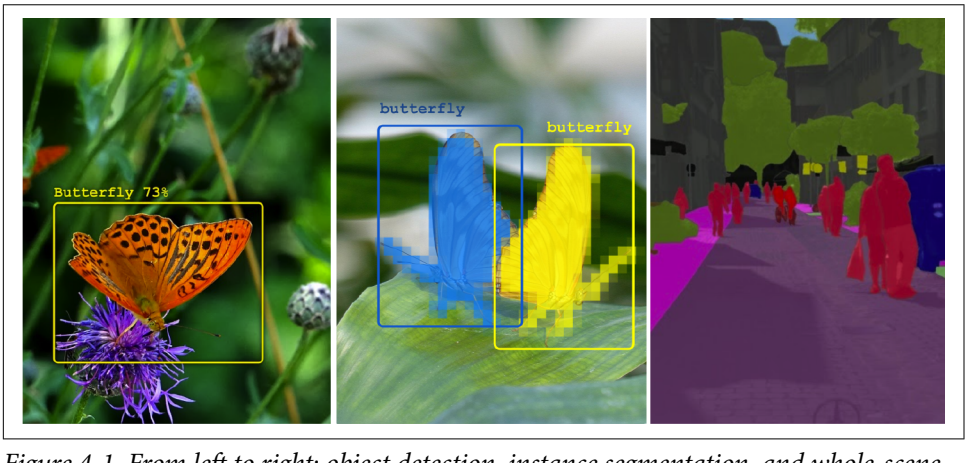


Figure 4-1. From left to right: object detection, instance segmentation, and whole-scene semantic segmentation. Images from Arthropods and Cityscapes datasets.

The code for this chapter is in the *04_detect_segment* folder of the book's GitHub repository. We will provide file names for code samples and notebooks where applicable.

Object Detection

Seeing is, for most of us, so effortless that, as we glimpse a butterfly from the corner of our eye and turn our head to enjoy its beauty, we don't even think about the millions of visual cells and neurons at play, capturing light, decoding the signals, and processing them into higher and higher levels of abstraction.

We saw in Chapter 3 how image recognition in ML works. However, the models presented in that chapter were built to classify an image as whole—they could not tell us where in the image a flower was. In this section, we will look at ways to build ML models that can provide this location information. This is a task known as *object detection* (Figure 4-2).

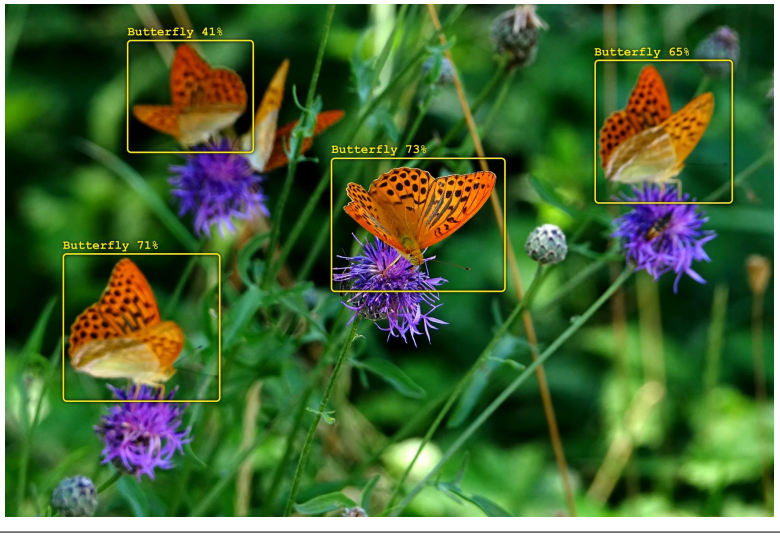


Figure 4-2. An object detection task. Image from Arthropods dataset.

In fact, convolutional layers do identify and locate the things they detect. The convolutional backbones from Chapter 3 already extract some location information. But in classification problems, the networks make no use of this information. They are trained on an objective where location does not matter. A picture of a butterfly is classified as such wherever the butterfly appears in the image. On the contrary, for object detection, we will add elements to the convolutional stack to extract and refine the location information and train the network to do so with maximum accuracy.

The simplest approach is to add something to the end of a convolutional backbone to predict bounding boxes around detected objects. That's the YOLO (You Only Look Once) approach, and we will start there. However, a lot of important information is also contained at intermediate levels in the convolutional backbone. To extract it, we

will build more complex architectures called feature pyramid networks (FPNs) and illustrate their use with RetinaNet.

In this section, we will be using the Arthropod Taxonomy Orders Object Detection dataset (Arthropods for short), which is freely available on Kaggle.com. The dataset contains seven categories—Coleoptera (beetles), Aranea (spiders), Hemiptera (true bugs), Diptera (flies), Lepidoptera (butterflies), Hymenoptera (bees, wasps, and ants), and Odonata (dragonflies)—as well as bounding boxes. Some examples are shown in Figure 4-3.

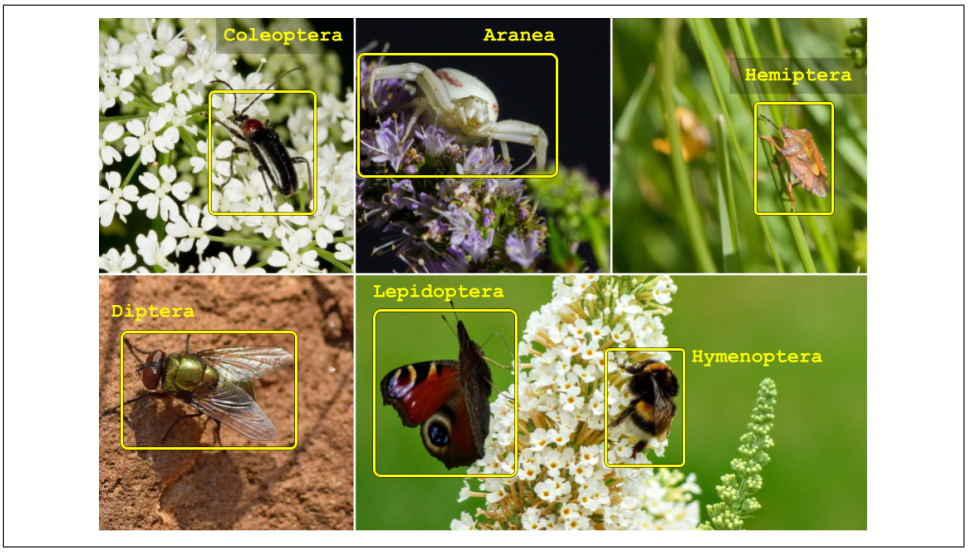


Figure 4-3. Some examples from the Arthropods dataset for object detection.

Besides YOLO, this chapter will also address the RetinaNet and Mask R-CNN architectures. Their implementations can be found in the TensorFlow Model Garden's official vision repository. We will be using the new implementations located, at the time of writing, in the "beta" folder of the repository.

Example code showing how to apply these detection models on a custom dataset such as Arthropods can be found in *04_detect_segment* on GitHub, in the folder corresponding to Chapter 4.

In addition to the TensorFlow Model Garden, there is also an excellent step-by-step implementation of RetinaNet on the keras.io website.

YOLO

YOLO (you only look once) is the simplest object detection architecture. It is not the most accurate, but it's one of the fastest when it comes to prediction times. For that

reason, it is used in many real-time systems like security cameras. The architecture can be based on any convolutional backbone from Chapter 3. Images are processed through the convolutional stack as in the image classification case, but the classification head is replaced with an object detection and classification head.

More recent variations of the YOLO architecture exist (YOLOv2, YOLOv3, YOLOv4), but we will not be covering them here. We will use YOLOv1 as our first stepping-stone into object detection architectures, because it is the simplest one to understand.

YOLO grid

YOLOv1 (hereafter referred to as "YOLO" for simplicity) divides a picture into a grid of *N*x*M* cells—for example, 7x5 (Figure 4-4). For each cell, it tries to predict a bounding box for an object that would be centered in that cell. The predicted bounding box can be larger than the cell from which it originates; the only constraint is that the center of the box is somewhere inside the cell.

What does it mean to predict a bounding box? Let's take a look.

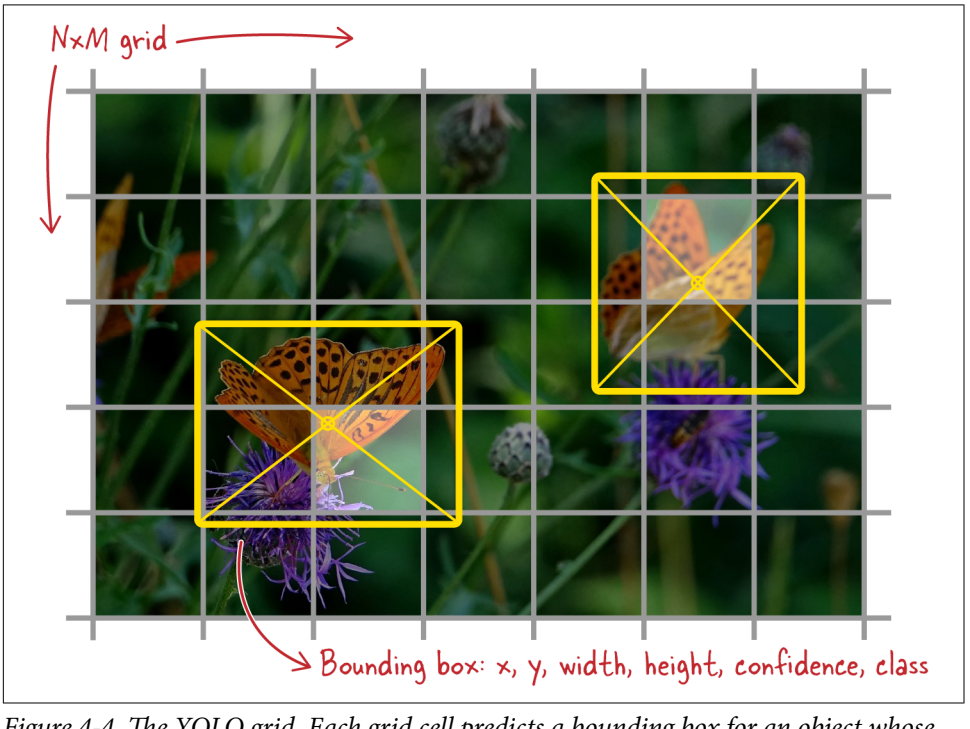


Figure 4-4. The YOLO grid. Each grid cell predicts a bounding box for an object whose center is somewhere in that cell. Image from Arthropods dataset.

Object detection head

Predicting a bounding box amounts to predicting six numbers: the four coordinates of the bounding box (in this case, the x and y coordinates of the center, and the width and height), a confidence factor which tells us if an object has been detected or not, and finally, the class of the object (for example, "butterfly"). The YOLO architecture does this directly on the last feature map, as generated by the convolutional backbone it is using.

In Figure 4-5, the x- and y-coordinate calculations use a hyperbolic tangent (tanh) activation so that the coordinates fall in the [–1, 1] range. They will be the coordinates of the center of the detection box, relative to the center of the grid cell they belong to.

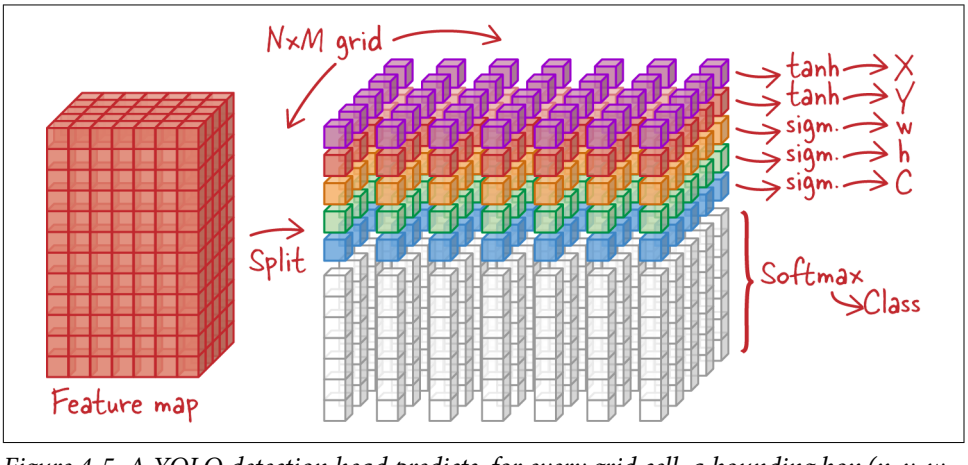


Figure 4-5. A YOLO detection head predicts, for every grid cell, a bounding box (x, y, w, h), the confidence C of there being an object in this location, and the class of the object.

Width and height (w, h) calculations use a sigmoid activation so as to fall in the [0, 1] range. They will represent the size of the detection box relative to the entire image. This allows detection boxes to be bigger than the grid cell they originate in. The confidence factor, C, is also in the [0, 1] range. Finally, a softmax activation is used to predict the class of the detected object. The tanh and sigmoid functions are depicted in Figure 4-6.

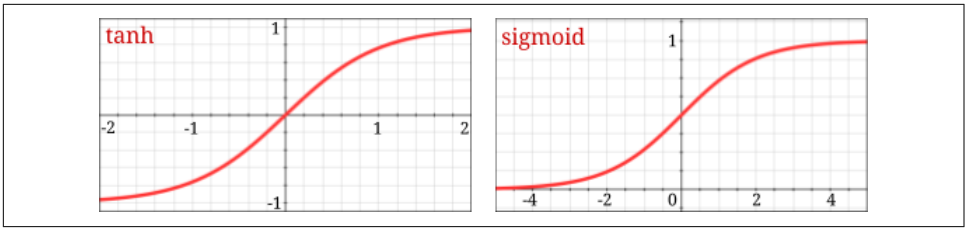


Figure 4-6. The tanh and sigmoid activation functions. Tanh outputs values in the [–1, 1] range, while the sigmoid function outputs them in the [0, 1] range.

An interesting practical question is how to obtain a feature map of exactly the right dimensions. In the example from Figure 4-4, it must contain exactly 7 * 5 * (5 + 7) values. The 7 * 5 is because we chose a 7x5 YOLO grid. Then, for each grid cell, five values are needed to predict a box (x, y, w, h, C), and seven additional values are needed because, in this example, we want to classify arthropods into seven categories (Coleoptera, Aranea, Hemiptera, Diptera, Lepidoptera, Hymenoptera, Odonata).

If you control the convolutional stack, you could try to tune it to get exactly 7 * 5 * 12 (420) outputs at the end. However, there is an easier way: flatten whatever feature map the convolutional backbone is returning and feed it through a fully connected layer with exactly that number of outputs. You can then reshape the 420 values into a 7x5x12 grid, and apply the appropriate activations as in Figure 4-5. The authors of the YOLO paper argue that the fully connected layer actually adds to the accuracy of the system.

Loss function

In object detection, as in any supervised learning setting, the correct answers are provided in the training data: ground truth boxes and their classes. During training the network predicts detection boxes, and it has to take into account errors in the boxes' locations and dimensions as well as misclassification errors, and also penalize detections of objects where there aren't any. The first step, though, is to correctly pair ground truth boxes with predicted boxes so that they can be compared. In the YOLO architecture, if each grid cell predicts a single box, this is straightforward. A ground truth box and a predicted box are paired if they are centered in the same grid cell (see Figure 4-4 for easier understanding).

However in the YOLO architecture, the number of detection boxes per grid cell is a parameter. It can be more than one. If you look back to Figure 4-5, you can see that it's easy enough for each grid cell to predict 10 or 15 (x, y, w, h, C) coordinates instead of 5 and generate 2 or 3 detection boxes instead of 1. But pairing these predictions with ground truth boxes requires more care. This is done by computing the *intersection over union* (IOU; see Figure 4-7) between all ground truth boxes and all predicted boxes within a grid cell, and selecting the pairings where the IOU is the highest.

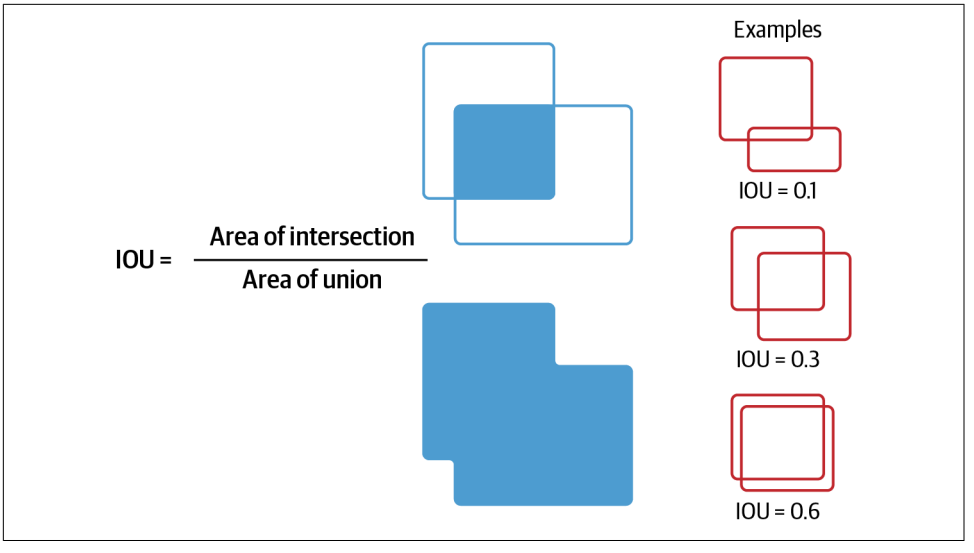


Figure 4-7. The IOU metric.

To summarize, ground truth boxes are assigned to grid cells by their centers and to the prediction boxes within these grid cells by IOU. With the pairings in place, we can now calculate the different parts of the loss:

Object presence loss
: Each grid cell that has a ground truth box computes:

$$L_{obj} = (1 - C)^2$$

Object absence loss
: Each grid cell that does not have a ground truth box computes:

$$L_{noobj} = \left(0 - C\right)^2 = C^2$$

Object classification loss
: Each grid cell that has a ground truth box computes:

$$L_{class} = cross\_entropy\left(p, \hat{p}\right)$$

where $\hat{p}$ is the vector of predicted class probabilities and p is the one-hot-encoded target class.

Bounding box loss

Each predicted box/ground truth box pairing contributes (predicted coordinate marked with a hat, the other coordinate is the ground truth):

$$L_{box} = (x - \hat{x})^2 + (y - \hat{y})^2 + (\sqrt{w} - \sqrt{\hat{w}})^2 + \left(\sqrt{h} - \sqrt{\hat{h}}\right)^2$$

Notice here that the difference in box sizes is computed on the square roots of the dimensions. This is to mitigate the effect of large boxes, which tend to overwhelm the loss.

Finally, all the loss contributions from the grid cells are added together, with weighting factors. A common problem in object detection losses is that small losses from numerous cells with no object in them end up overpowering the loss from a lone cell that predicts a useful box. Weighting different parts of the loss can alleviate this problem. The authors of the paper used the following empirical weights:

$$\lambda_{obj} = 1 \;\; \lambda_{noobj} = 0.5 \;\; \lambda_{class} = 1 \;\; \lambda_{box} = 5$$

YOLO limitations

The biggest limitation is that YOLO predicts a single class per grid cell and will not work well if multiple objects of different kinds are present in the same cell.

The second limitation is the grid itself: a fixed grid resolution imposes strong spatial constraints on what the model can do. YOLO models will typically not do well on collections of small objects, like a flock of birds, without careful tuning of the grid to the dataset.

Also, YOLO tends to localize objects with relatively low precision. The main reason for that is that it works on the last feature map from the convolutional stack, which is typically the one with the lowest spatial resolution and contains only coarse location signals.

Despite these limitations, the YOLO architecture is very simple to implement, especially with a single detection box per grid cell, which makes it a good choice when you want to experiment with your own code.

Note that it is not the case that every object is detected by looking at the information in a single grid cell. In a sufficiently deep convolutional neural network (CNN), every value in the last feature map, from which detection boxes are computed, depends on all the pixels of the original image.

If a higher accuracy is needed, you can step up to the next level: RetinaNet. It incorporates a number of ideas that improve upon the basic YOLO architecture, and is regarded, at the time of writing, as the state of the art of so-called *single-shot detectors*.

RetinaNet

RetinaNet, as compared to YOLOv1, has several innovations in its architecture and in the design of its losses. The neural network design includes feature pyramid networks which combine information extracted at multiple scales. The detection head predicts boxes starting from *anchor boxes* that change the bounding box representation to make training easier. Finally, the loss innovations include the focal loss, a loss specifically designed for detection problems, a smooth L1 loss for box regression, and non-max suppression. Let's look at each of these in turn.

Feature pyramid networks

When an image is processed by a CNN, the initial convolutional layers pick up low-level details like edges and textures. Further layers combine them into features with more and more semantic value. At the same time, pooling layers in the network reduce the spatial resolution of the feature maps (see Figure 4-8).

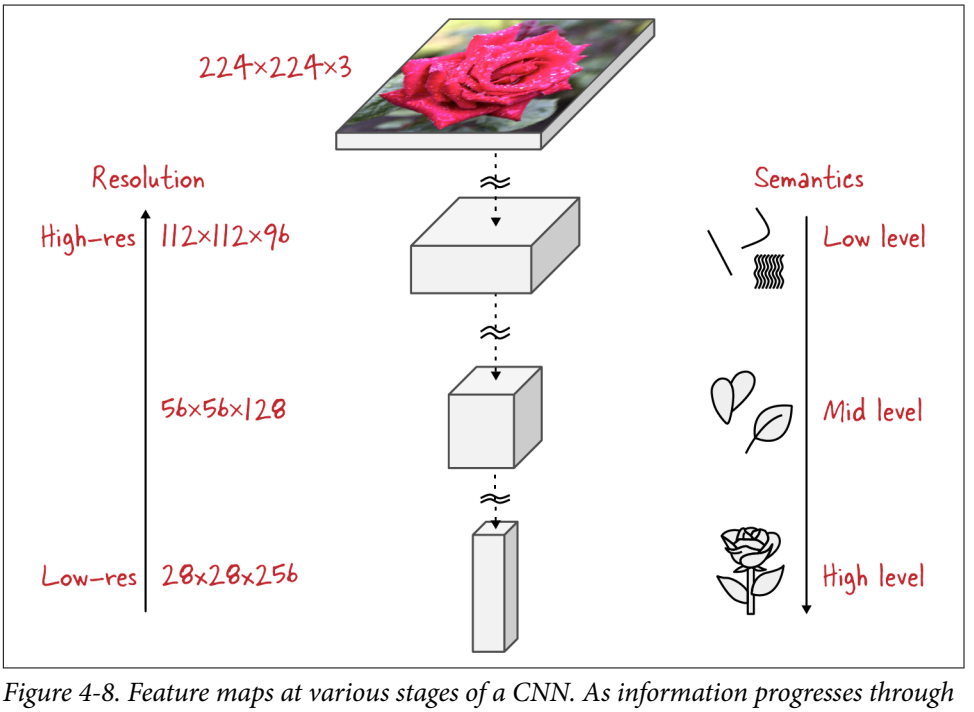


Figure 4-8. Feature maps at various stages of a CNN. As information progresses through the neural network, its spatial resolution decreases but its semantic content increases from low-level details to high-level objects.

The YOLO architecture only uses the last feature map for detection. It is able to correctly identify objects, but its localization accuracy is limited. Another idea would be to try and add a detection head at every stage. Unfortunately, in this approach, the heads working from the early feature maps would localize objects rather well but would have difficulty labeling them. At that early stage, the image has only gone through a couple of convolutional layers, which is not enough to classify it. Higher-level semantic information, like "this is a rose," needs tens of convolutional layers to emerge.

Still, one popular detection architecture, called the single-shot detector (SSD), is based on this idea. The authors of the SSD paper made it work by connecting their multiple detection heads to multiple feature maps, all located toward the end of the convolutional stack.

What if we could combine all feature maps in a way that would surface both good spatial information and good semantic information at all scales? This can be done with a couple of additional layers forming a feature pyramid network. Figure 4-9 offers a schematic view of an FPN compared to the YOLO and SSD approaches, while Figure 4-10 presents the detailed design.

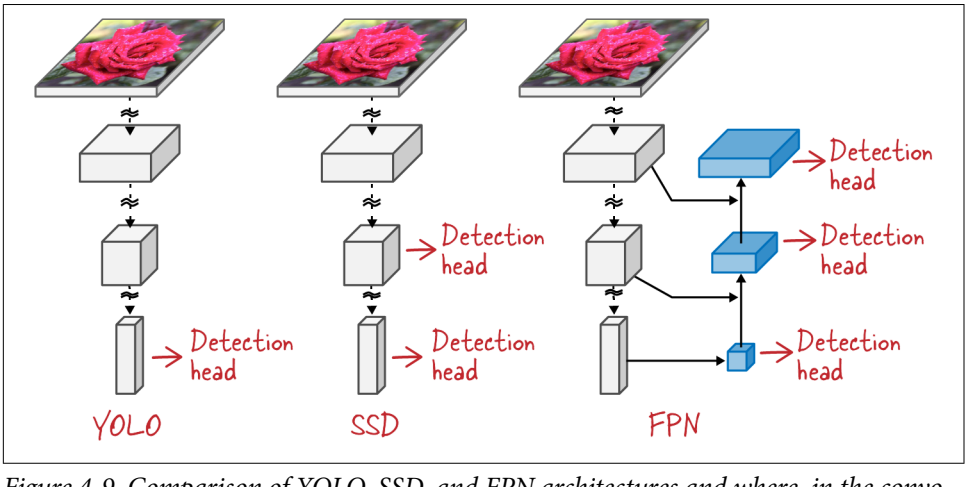


Figure 4-9. Comparison of YOLO, SSD, and FPN architectures and where, in the convolutional stack, they connect their detection head(s).

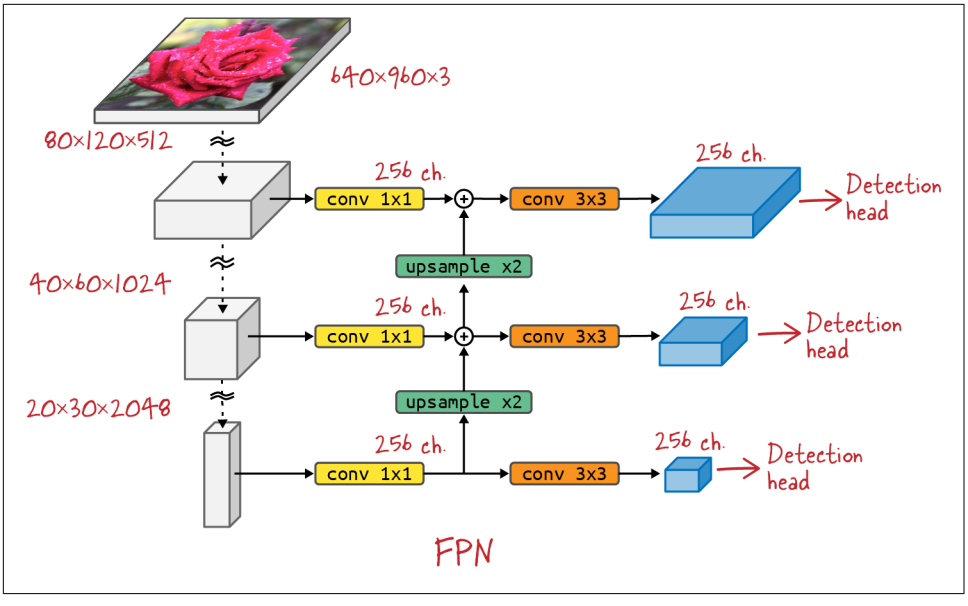


Figure 4-10. A feature pyramid network in detail. Feature maps are extracted from various stages of a convolutional backbone, and 1x1 convolutions squeeze every feature map to the same number of channels. Upsampling (nearest neighbor) then makes their spatial dimensions compatible so that they can be added up. The final 3x3 convolutions smooth out upsampling artifacts. Typically no activation functions are used in the FPN layers.

Here is what is happening in the FPN in Figure 4-10: in the downward path (convolutional backbone), convolutional layers gradually refine the semantic information in the feature maps, while pooling layers scale the feature maps down in their spatial dimensions (the *x* and *y* dimensions of the image). In the upward path, feature maps from the bottom layers containing good high-level semantic information get upsampled (using a simple nearest neighbor algorithm) so that they can be added, elementwise, to feature maps higher up in the stack. 1x1 convolutions are used in the lateral connections to bring all feature maps to the same channel depth and make the additions possible. The FPN paper, for example, uses 256 channels everywhere. The resulting feature maps now contain semantic information at all scales, which was the initial goal. They are further processed through a 3x3 convolution, mostly to smooth out the effects of the upsampling.

There are typically no nonlinearities in the FPN layers. The authors of the FPN paper found them to have little impact.

A detection head can now take the feature maps at each resolution and produce box detections and classifications. The detection head can itself have multiple designs, which we will cover in the next two sections. It will, however, be shared across all the

feature maps at different scales. This is why it was important to bring all the feature maps to the same channel depth.

The nice thing about the FPN design is that it is independent of the underlying convolutional backbone. Any convolutional stack from Chapter 3 will do, as long as you can extract intermediate feature maps from it—typically four to six, at various scales. You can even use a pretrained backbone. Typical choices are ResNet or EfficientNet, and pretrained versions of them can be found in TensorFlow Hub.

There are multiple levels in a convolutional stack where features can be extracted and fed into the FPN. For each desired scale, many layers output feature maps of the same dimensions (see Figure 3-26 in the previous chapter). The best choice is the last feature map of a given block of layers outputting similarly sized features, just before a pooling layer halves the resolution again. This feature map is likely to contain the strongest semantic features.

It is also possible to extend an existing pretrained backbone with additional pooling and convolutional layers, for the sole purpose of feeding an FPN. These additional feature maps are typically small and therefore fast to process. They correspond to the lowest spatial resolution (see Figure 4-8) and can therefore improve the detection of large objects. The SSD paper actually used this trick, and RetinaNet does as well, as you will see in the architecture diagram later (Figure 4-15).

Anchor boxes

In the YOLO architecture, detection boxes are computed as deltas relative to a set of base boxes ($\Delta x = x - x_0$, $\Delta y = y - y_0$, $\Delta w = w - w_0$, $\Delta h = h - h_0$ are often referred to as "deltas" relative to some base box x_0, y_0, w_0, h_0 because of the Greek letter Δ, usually chosen to represent a "difference"). In that case, the base boxes were a simple grid overlaid on the image (see Figure 4-4).

More recent architectures have expanded on this idea by explicitly defining a set of so-called "anchor boxes" with various aspect ratios and scales (examples in Figure 4-11). Predictions are again small variations of the size and position of the anchors. The goal is to help the neural network predict small values around zero rather than large ones. Indeed, neural networks are able to solve complex nonlinear problems because they use nonlinear activation functions between their layers. However, most activation functions (sigmoid, ReLU) exhibit a nonlinear behavior around zero only. That's why neural networks are at their best when they predict small values around zero, and it's why predicting detections as small deltas relative to anchor boxes is helpful. Of course, this only works if there are enough anchor boxes of various sizes and aspect ratios that any object detection box can be paired (by max IOU) with an anchor box of closely matching position and dimensions.

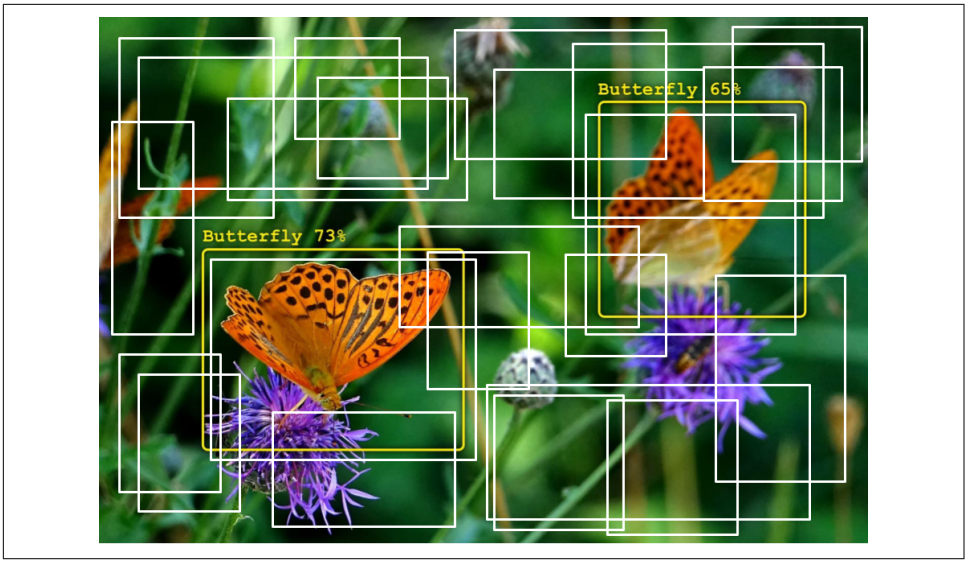


Figure 4-11. Examples of anchor boxes of various sizes and aspect ratios used to predict detection boxes. Image from Arthropods dataset.

We will describe in detail the approach taken in the RetinaNet architecture, as an example. RetinaNet uses nine different anchor types with:

- Three different aspect ratios: 2:1, 1:1, 1:2
- Three different sizes: 2^0, $2^{⅓}$, $2^{⅔}$ ($\simeq$ 1, 1.3, 1.6)

They are depicted in Figure 4-12.

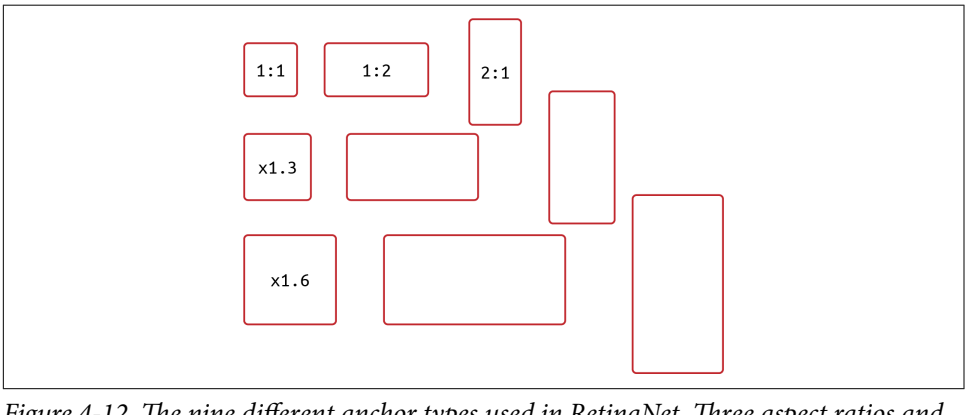


Figure 4-12. The nine different anchor types used in RetinaNet. Three aspect ratios and three different sizes.

Anchors, along with the feature maps computed by an FPN, are the inputs from which detections are computed in RetinaNet. The sequence of operations is as follows:

- The FPN reduces the input image into five feature maps (see Figure 4-10).
- Each feature map is used to predict bounding boxes relative to anchors at regularly spaced locations throughout the image. For example, a feature map of size 4x6 with 256 channels will use 24 (4 * 6) anchor locations in the image (see Figure 4-13).
- The detection head uses multiple convolutional layers to convert the 256-channel feature map into exactly 9 * 4 = 36 channels, yielding 9 detection boxes per location. The four numbers per detection box represent the deltas relative to the center (x, y), the width, and the height of the anchor. The precise sequence of the layers that compute detections from the feature maps is shown in Figure 4-15.
- Finally, each feature map from the FPN, since it corresponds to a different scale in the image, will use different scales of anchor boxes.

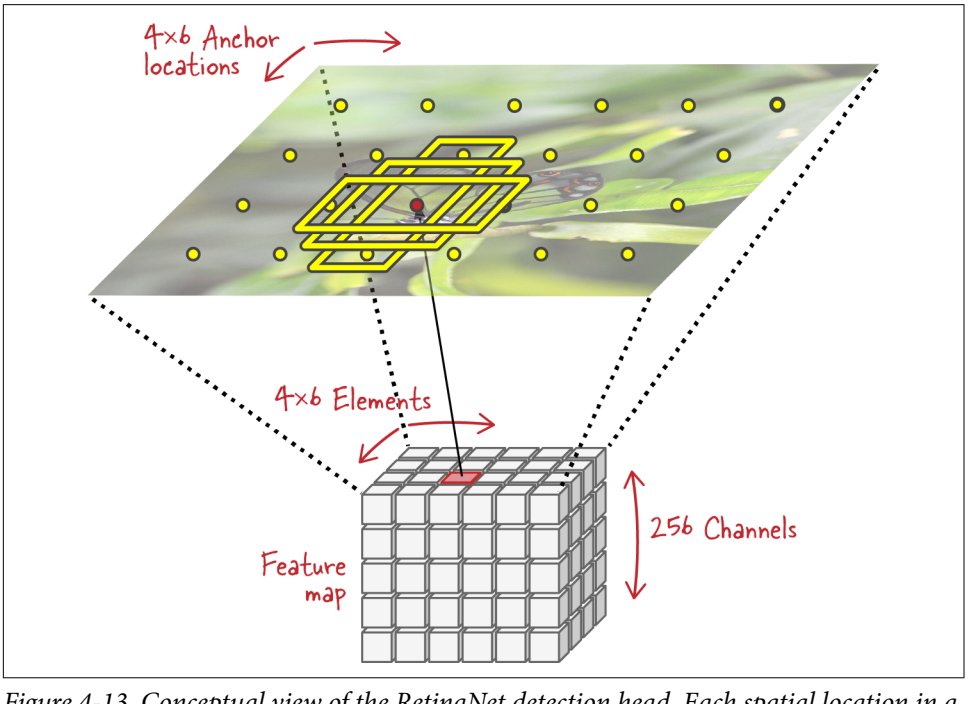


Figure 4-13. Conceptual view of the RetinaNet detection head. Each spatial location in a feature map corresponds to a series of anchors in the image, all centered at the same point. For clarity, only three such anchors are shown in the illustration, but RetinaNet would have nine at every location.

The anchors themselves are spaced regularly across the input image and sized appropriately for each level of the feature pyramid. For example in RetinaNet, the following parameters are used:

- The feature pyramid has five levels corresponding to scales P_3, P_4, P_5, P_6, and P_7 in the backbone. Scale P_n represents a feature map 2^n times smaller in width and height than the input image (see the complete RetinaNet view in Figure 4-15).
- Anchor base sizes are 32x32, 64x64, 128x128, 256x256, 512x512 pixels, at each feature pyramid level respectively (= $4 * 2^n$, if n is the scale level).
- Anchor boxes are considered for every spatial location of every feature map in the feature pyramid, which means that the boxes are spaced every 8, 16, 32, 64, or 128 pixels across the input image at each feature pyramid level, respectively (= 2^n, if n is the scale level).

The smallest anchor box is therefore 32x32 pixels while the largest one is 812x1,624 pixels.

The anchor box settings must be tuned for every dataset so that they correspond to the detection box characteristics actually found in the training data. This is typically done by resizing input images rather than changing the anchor box generation parameters. However, on specific datasets with many small detections, or, on the contrary, mostly large objects, it can be necessary to tune the anchor box generation parameters directly.

The last step is to compute a detection loss. For that, predicted detection boxes must be paired with ground truth boxes so that detection errors can be evaluated.

The assignment of ground truth boxes to anchor boxes is based on the IOU metric computed between each set of boxes in one input image. All pairwise IOUs are computed and are arranged in a matrix with N rows and M columns, N being the number of ground truth boxes and M the number of anchor boxes. The matrix is then analyzed by columns (see Figure 4-14):

- An anchor is assigned to the ground truth box that has the largest IOU in its column, provided it is more than 0.5.
- An anchor box that has no IOU greater than 0.4 in its column is assigned to detect nothing (i.e., the background of the image).
- Any unassigned anchor at this point is marked to be ignored during training. Those are anchors with IOUs in the intermediate regions between 0.4 and 0.5.

Now that every ground truth box is paired with exactly one anchor box, it is possible to compute box predictions, classifications, and the corresponding losses.

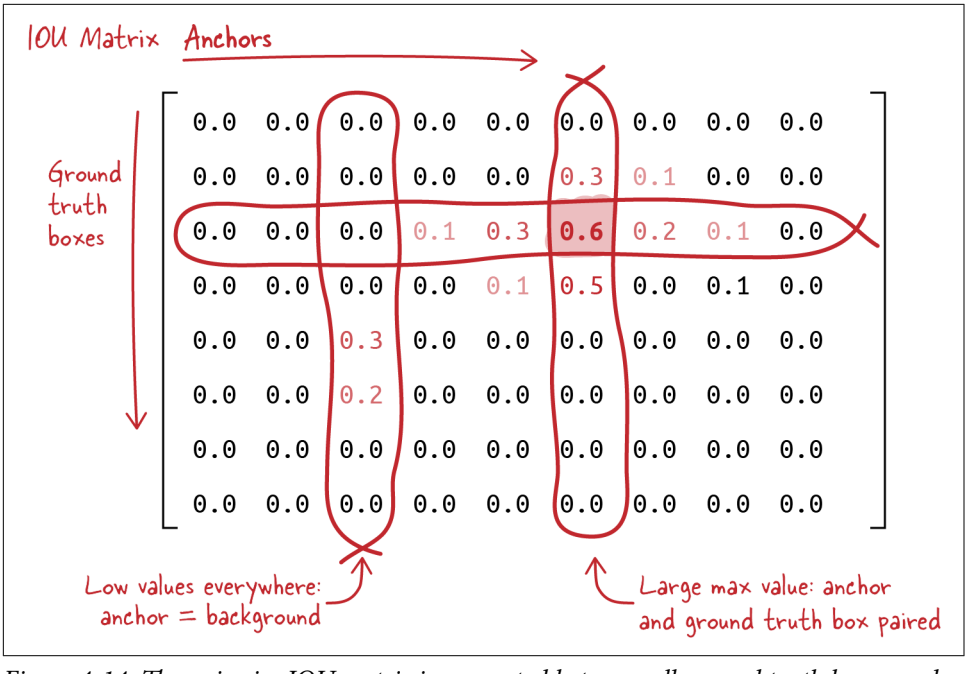


Figure 4-14. The pairwise IOU metric is computed between all ground truth boxes and all anchor boxes to determine their pairings. Anchors without a meaningful intersection with a ground truth box are deemed "background" and trained to detect nothing.

Architecture

The detection and classification heads transform the feature maps from the FPN into class predictions and bounding box deltas. Feature maps are three-dimensional. Two of their dimensions correspond to the x and y dimensions of the image and are called *spatial dimensions*; the third dimension is their number of channels.

In RetinaNet, for every spatial location in every feature map, the following parameters are predicted (with K = the number of classes and B = the number of anchor box types, so in our case B=9):

- The class prediction head predicts $B * K$ probabilities, one set of probabilities for every anchor type. This in effect predicts one class for every anchor.
- The detection head predicts $B * 4 = 36$ box deltas Δx, Δy, Δw, Δh. Bounding boxes are still parameterized by their center (x, y) as well as their width and height (w, h).

Both heads share a similar design, although with different weights, and the weights are shared across all scales in the feature pyramid.

Figure 4-15 represents a complete view of the RetinaNet architecture. It uses a ResNet50 (or other) backbone. The FPN extracts features from backbone levels P_3 though P_7, where P_n is the level where the feature map is reduced by a factor of 2^n in its width and height compared to the original image. The FPN part is described in detail in Figure 4-10. Every feature map from the FPN is fed through both a classification and a box regression head.

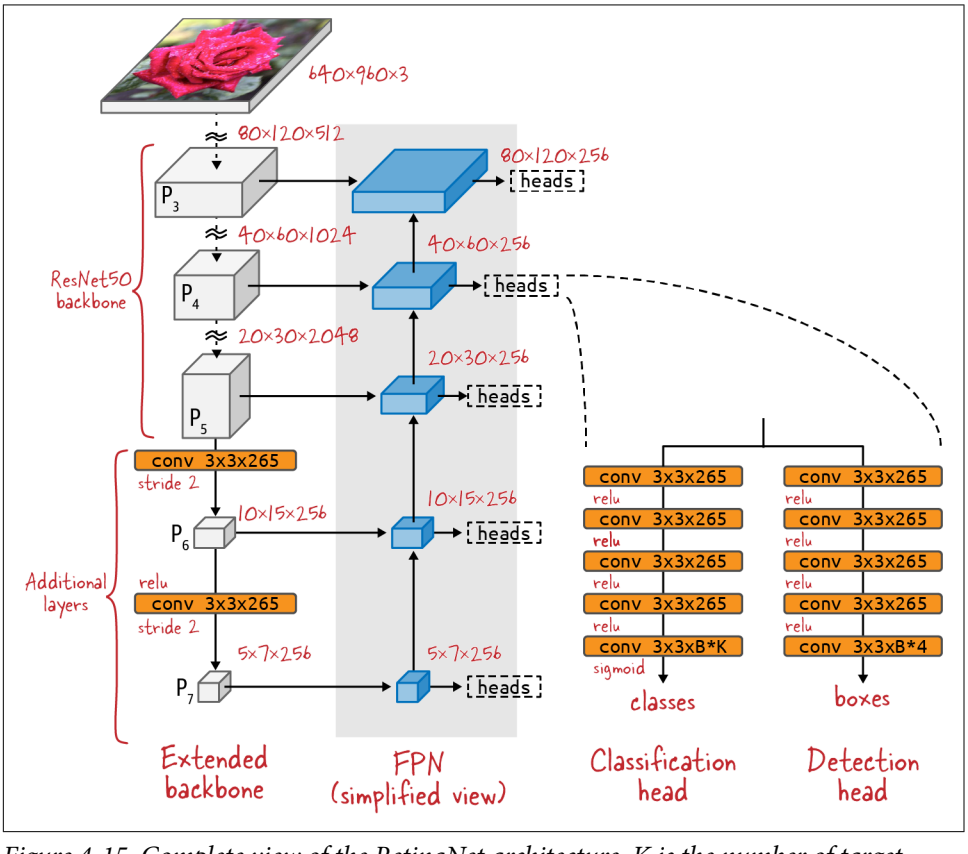


Figure 4-15. Complete view of the RetinaNet architecture. K is the number of target classes. B is the number of anchor boxes at each position, which is nine in RetinaNet.

The RetinaNet FPN taps into the three last scale levels available from the backbone. The backbone is extended with 2 additional layers using a stride of 2 to provide 2 additional scale levels to the FPN. This architectural choice allows RetinaNet to avoid processing very large feature maps, which would be time-consuming. The addition of the last two coarse scale levels also improves the detection of very large objects.

The classification and box regression heads themselves are made from a simple sequence of 3x3 convolutions. The classification head is designed to predict *K* binary

classifications for every anchor, which is why it ends on a sigmoid activation. It looks like we are allowing multiple labels to be predicted for every anchor, but actually the goal is to allow the classification head to output all zeros, which will represent the "background class" corresponding to no detections. A more typical activation for classification would be softmax, but the softmax function cannot output all zeros.

The box regression ends with no activation function. It is computing the differences between the center coordinates (*x*, *y*), width, and height of the anchor box and detection box. Some care must be taken to allow the regressor to work in the [–1, 1] range at all levels in the feature pyramid. The following formulas are used to achieve that:

- $X_{pixels} = X \times U \times W_A + X_A$
- $Y_{pixels} = Y \times U \times H_A + Y_A$
- $W_{pixels} = W_A \times e^{W \times V}$
- $H_{pixels} = H_A \times e^{H \times V}$

In these formulas, X_A, Y_A, W_A, and H_A are the coordinates of an anchor box (center coordinates, width, height), while X, Y, W, and H are the predicted coordinates relative to the anchor box (deltas). X_{pixels}, Y_{pixels}, W_{pixels}, and H_{pixels} are the actual coordinates, in pixels, of the predicted box (center and size). U and V are modulating factors that correspond to the expected variance of the deltas relative to the anchor box. Typical values are U=0.1 for coordinates, and V=0.2 for sizes. You can verify that values in the [–1, 1] range for predictions result in predicted boxes that fall within ±10% of the position of the anchor and within ±20% of its size.

Focal loss (for classification)

How many anchor boxes are considered for one input image? Looking back at Figure 4-15, with an example input image of 640x960 pixels, the five different feature maps in the feature pyramid represent 80 * 120 + 40 * 60 + 20 * 30+ 10 * 15 + 5 * 7 = 12,785 locations in the input image. With 9 anchor boxes per location, that's slightly over 100K anchor boxes.

This means that 100K predicted boxes will be generated for every input image. In comparison, there are 0 to 20 ground truth boxes per image in a typical application. The problem this creates in detection models is that the loss corresponding to background boxes (boxes assigned to detect nothing) can overwhelm the loss corresponding to useful detections in the total loss. This happens even if background detections are already well trained and produce a small loss. This small value multiplied by 100K can still be orders of magnitude larger than the detection loss for actual detections. The end result is a model that cannot be trained.

The RetinaNet paper suggested an elegant solution to this problem: the authors tweaked the loss function to produce much smaller values on empty backgrounds. They call this the *focal loss*. Here are the details.

We have already seen that RetinaNet uses a sigmoid activation to generate class probabilities. The output is a series of binary classifications, one for every class. A probability of 0 for every class means "background"; i.e., nothing to detect here. The classification loss used is the binary cross-entropy. For every class, it is computed from the actual binary class label y (0 or 1) and the predicted probability for the class p using the following formula:

$$CE(y, p) = -y \cdot \log(p) - (1 - y) \cdot \log(1 - p)$$

The focal loss is the same formula with a small modification:

$$FL(y, p) = -y \cdot (1 - p)^{\gamma} \cdot \log(p) - (1 - y) \cdot p^{\gamma} \cdot \log(1 - p)$$

For γ=0 this is exactly the binary cross-entropy, but for higher values of γ the behavior is slightly different. To simplify, let's only consider the case of background boxes that do not belong to any class (i.e., where y=0 for all classes):

$$FL_{bkg}(p) = -p^{\gamma} \cdot \log(1 - p)$$

And let's plot the values of the focal loss for various values of p and γ (Figure 4-16).

As you can see in the figure, with γ=2, which was found to be an adequate value, the focal loss is much smaller than the regular cross-entropy loss, especially for small values of p. For background boxes, where there is nothing to detect, the network will quickly learn to produce small class probabilities p across all classes. With the cross-entropy loss, these boxes, even well classified as "background" with p=0.1 for example, would still be contributing a significant amount: CE(0.1) = 0.05. The focal loss is 100 times less: FL(0.1) = 0.0005.

With the focal loss, it becomes possible to add the losses from all anchor boxes—all 100K of them—and not worry about the total loss being overwhelmed by thousands of small losses from easy-to-classify background boxes.

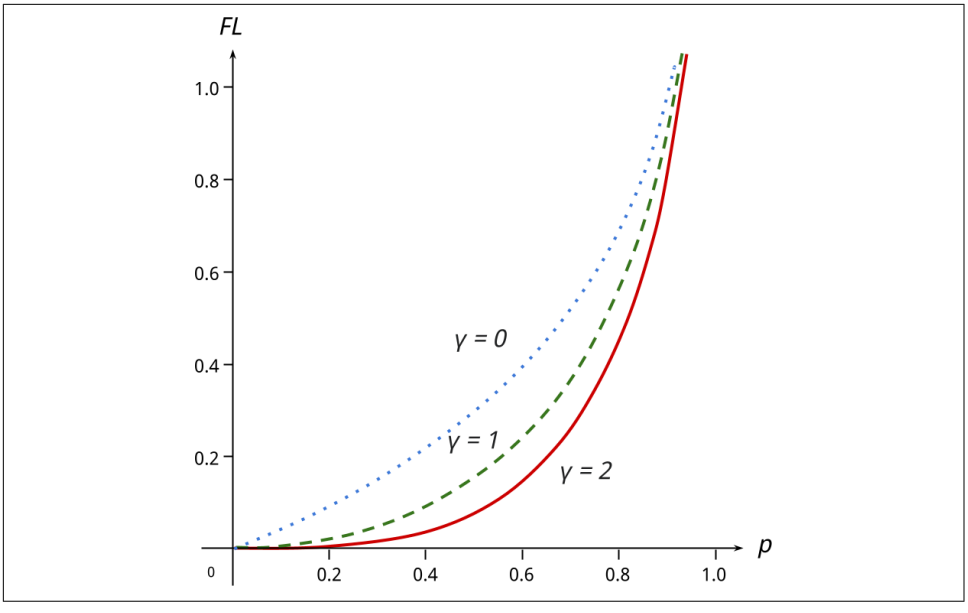


Figure 4-16. Focal loss for various values of γ. For γ=0, this is the cross-entropy loss. For higher values of γ, the focal loss greatly de-emphasizes easy-to-classify background regions where p is close to 0 for every class.

Smooth L1 loss (for box regression)

Detection boxes are computed by a regression. For regressions, the two most common losses are L1 and L2, also called *absolute loss* and *squared loss*. Their formulas are (computed between a target value a and the predicted value $\hat{a}$):

$$L1(a, \hat{a}) = |a - \hat{a}|$$
$$L2(a, \hat{a}) = (a - \hat{a})^2$$

The problem with the L1 loss is that its gradient is the same everywhere, which is not great for learning. The L2 loss is therefore preferred for regressions—but it suffers from a different problem. In the L2 loss, differences between the predicted and target values are squared, which means that the loss tends to get very large as the prediction and the target grow apart. This becomes problematic if you have some outliers, like a couple of bad points in the data (for example, a target box with the wrong size). The result will be that the network will try to fit the bad data point at the expense of everything else, which is not good either.

A good compromise between the two is the *Huber loss,* or *smooth L1 loss* (see Figure 4-17). It behaves like the L2 loss for small values and like the L1 loss for large values. Close to zero, it has the nice property that its gradient is larger when the

differences are larger, and therefore it pushes the network to learn more where it is making the biggest mistakes. For large values, it becomes linear instead of quadratic and avoids being thrown off by a couple of bad target values. Its formula is:

$$L_\delta(a - \hat{a}) = \frac{1}{2}(a - \hat{a})^2 \quad \text{for} \left|a - \hat{a}\right| \leq \delta$$

$$L_\delta = \delta\left(\left|a - \hat{a}\right| - \frac{1}{2}\delta\right) \quad \text{otherwise}$$

Where δ is an adjustable parameter. δ is the value around which the behavior switches from quadratic to linear. Another formula can be used to avoid the piecewise definition:

$$L_\delta(a - \hat{a}) = \delta^2\left(\sqrt{1 + \left(\frac{a - \hat{a}}{\delta}\right)^2} - 1\right)$$

This alternate form does not give the exact same values as the standard Huber loss, but it has the same behavior: quadratic for small values, linear for large ones. In practice, either form will work well in RetinaNet, with δ=1.

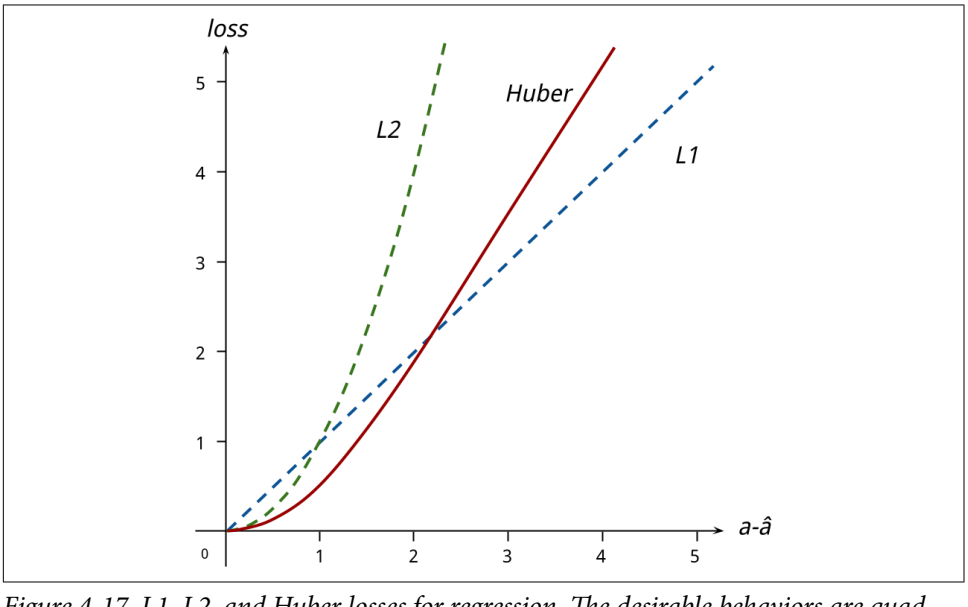


Figure 4-17. L1, L2, and Huber losses for regression. The desirable behaviors are quadratic for small values and linear for large ones. The Huber loss has both.

Non-maximum suppression

A detection network using numerous anchor boxes, such as RetinaNet, usually produces multiple candidate detections for every target box. We need an algorithm to select a single detection box for every detected object.

Non-maximum suppression (NMS) takes box overlap (IOU) and class confidence into account to select the most representative box for a given object (Figure 4-18).

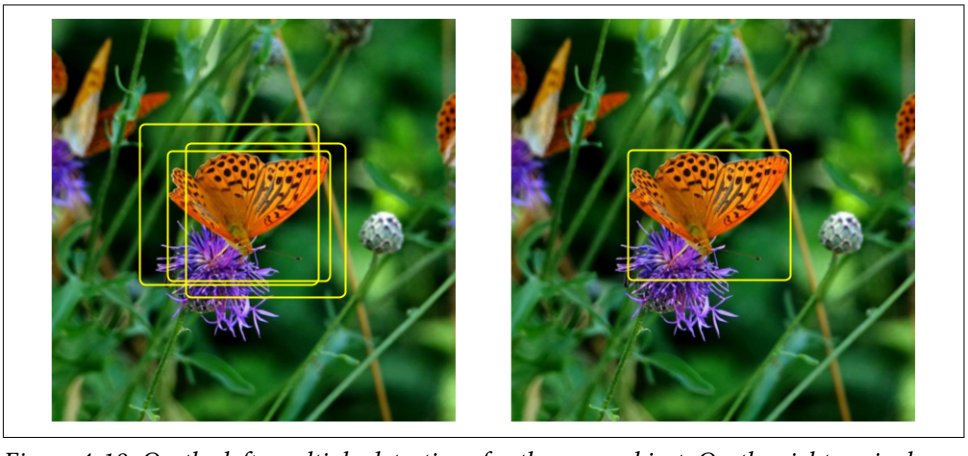


Figure 4-18. On the left: multiple detections for the same object. On the right: a single box remaining after non-max suppression. Image from Arthropods dataset.

The algorithm uses a simple "greedy" approach: for every class, it considers the overlap (IOU) between all the predicted boxes. If two boxes overlap more than a given value A (IOU > A), it keeps the one with the highest class confidence. In Python-like pseudocode, for one given class:

```
def NMS(boxes, class_confidence):
    result_boxes = []
    for b1 in boxes:
        discard = False
        for b2 in boxes:
            if IOU(b1, b2) > A:
                if class_confidence[b2] > class_confidence[b1]:
                    discard = True
        if not discard:
            result_boxes.append(b1)
    return result_boxes
```

NMS works quite well in practice but it can have some unwanted side effects. Notice that the algorithm relies on a single threshold value (A). Changing this value changes the box filtering, especially for adjacent or overlapping objects in the original image. Take a look at the example in Figure 4-19. If the threshold is set at A=0.4, then the two boxes detected in the figure will be regarded as "overlapping" for the same class

and the one with the lowest class confidence (the one on the left) will be discarded. That is obviously wrong. There are two butterflies to detect in this image and, before NMS, both were detected with a high confidence.

Figure 4-19. Objects close to each other create a problem for the non-max suppression algorithm. If the NMS threshold is 0.4, the box detected on the left will be discarded, which is wrong. Image from Arthropods dataset.

Pushing the threshold value higher will help, but if it's too high the algorithm will fail to merge boxes that correspond to the same object. The usual value for this threshold is A=0.5, but it still causes objects that are close together to be detected as one.

A slight variation on the basic NMS algorithm is called *Soft-NMS*. Instead of removing non-maximum overlapping boxes altogether, it lowers their confidence score by the factor:

$$exp\left(-\frac{IOU^2}{\sigma}\right)$$

with σ being an adjustment factor that tunes the strength of the Soft-NMS algorithm. A typical value is σ=0.5. The algorithm is applied by considering the box with the highest confidence score for a given class (the *max box*), and decreasing the scores for all other boxes by this factor. The max box is then put aside and the operation is repeated on the remaining boxes until none remain.

For nonoverlapping boxes (IOU=0), this factor is 1. The confidence factors of boxes that do not overlap the max box are thus not affected. The factor gradually, but continuously, decreases as boxes overlap more with the max box. Highly overlapping boxes (IOU=0.9) get their confidence factor decreased by a lot (×0.2), which is the

expected behavior because they are redundant with the max box and we want to get rid of them.

Since the Soft-NMS algorithm does not discard any boxes, a second threshold, based on the class confidence, is used to actually prune the list of detections.

The effect of Soft-NMS on the example from Figure 4-19 is shown in Figure 4-20.

Figure 4-20. Objects close to each other as handled by Soft-NMS. The detection box on the left is not deleted, but its confidence factor is reduced from 78% to 55%. Image from Arthropods dataset.

In TensorFlow, both styles of non-max suppression are available. Standard NMS is called `tf.image.non_max_suppression`, while Soft-NMS is called `tf.image.non_max_suppression_with_scores`.

Other considerations

In order to reduce the amount of data needed, it is customary to use a pretrained backbone.

Classification datasets are much easier to put together than object detection datasets. That's why readily available classification datasets are typically much larger than object detection datasets. Using a pretrained backbone from a classifier allows you to combine a generic large classification dataset with a task-specific object detection dataset and obtain a better object detector.

The pretraining is done on a classification task. Then the classification head is removed and the FPN and detection heads are added, initialized at random. The actual object detection training is performed with all weights trainable, which means

that the backbone will be fine-tuned while the FPN and detection head train from scratch.

Since detection datasets tend to be smaller, data augmentation (which we will cover in more detail in Chapter 6) plays an important part in training. The basic data augmentation technique is to cut fixed-sized crops out of the training images at random, and at random zoom factors (see Figure 4-21). With target bounding boxes adjusted appropriately, this allows you to train the network with the same object at different locations in the image, at different scales and with different parts of the background visible.

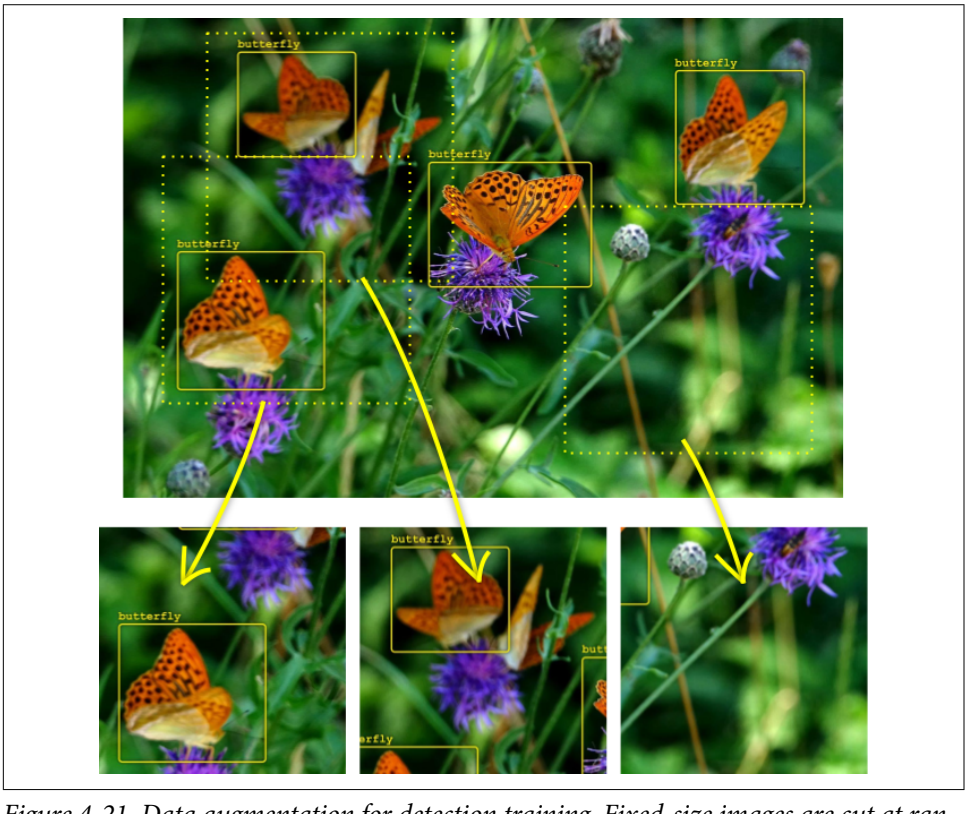


Figure 4-21. Data augmentation for detection training. Fixed-size images are cut at random from each training image, potentially at different zoom factors. Target box coordinates are recomputed relative to the new boundaries. This provides more training images and more object locations from the same initial training data. Image from Arthropods dataset.

A practical advantage of this technique is that it also provides fixed-sized training images to the neural network. You can train directly on a training dataset made up of

images of different sizes and aspect ratios. The data augmentation takes care of getting all the images to the same size.

Finally, what drives training and hyperparameter tuning are metrics. Object detection problems have been the subject of multiple large-scale contests where detection metrics have been carefully standardized; this topic is covered in detail in "Metrics for Object Detection" on page 297 in Chapter 8.

Now that we have looked at object detection, let's turn our attention to another class of problems: image segmentation.

Segmentation

Object detection finds bounding boxes around objects and classifies them. *Instance segmentation* adds, for every detected object, a pixel mask that gives the shape of the object. *Semantic segmentation*, on the other hand, does not detect specific instances of objects but classifies every pixel of the image into a category like "road," "sky," or "people."

Mask R-CNN and Instance Segmentation

YOLO and RetinaNet, which we covered in the previous section, are examples of single-shot detectors. An image traverses them only once to produce detections. Another approach is to use a first neural network to suggest potential locations for objects to be detected, then use a second network to classify and fine-tune the locations of these proposals. These architectures are called *region proposal networks* (RPNs).

They tend to be more complex and therefore slower than single-shot detectors, but are also more accurate. There is a long list of RPN variants, all based on the original "regions with CNN features" idea: R-CNN, Fast R-CNN, Faster R-CNN, and more. The state of the art, at the time of writing, is Mask R-CNN, and that's the architecture we are going to dive into next.

The main reason why it is important to be aware of architectures like Mask R-CNN is not their marginally superior accuracy, but the fact that they can be extended to perform instance segmentation tasks. In addition to predicting a bounding box around detected objects, they can be trained to predict their outline—i.e., find every pixel belonging to each detected object (Figure 4-22). Of course, training them remains a supervised training task and the training data will have to contain ground truth segmentation masks for all objects. Unfortunately, masks are more time-consuming to generate by hand than bounding boxes and therefore instance segmentation datasets are harder to find than simple object detection datasets.

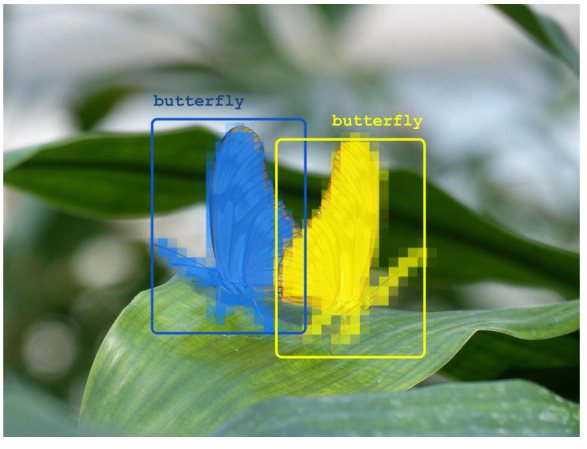


Figure 4-22. Instance segmentation involves detecting objects and finding all the pixels that belong to each object. The objects in the images are shaded with a pixel mask. Image from Arthropods dataset.

Let's look at RPNs in detail, first analyzing how they perform classic object detection, then how to extend them for instance segmentation.

Region proposal networks

An RPN is a simplified single-shot detection network that only cares about two classes: objects and background. An "object" is anything labeled as such in the dataset (any class), and "background" is the designated class for a box that does not contain an object.

An RPN can use an architecture similar to the RetinaNet setup we looked at earlier: a convolutional backbone, a feature pyramid network, a set of anchor boxes, and two heads. One head is for predicting boxes and the other is for classifying them as object or background (we are not predicting segmentation masks yet).

The RPN has its own loss function, computed from a slightly modified training dataset: the class of any ground truth object is replaced with a single class "object." The loss function used for boxes is, as in RetinaNet, the Huber loss. For classes, since this is a binary classification, binary cross-entropy is the best choice.

Boxes predicted by the RPN then undergo non-max suppression. The top N boxes, sorted by their probability of being an "object," are regarded as box proposals or *regions of interest* (ROIs) for the next stage. N is usually around one thousand, but if fast inference is important, it can be as little as 50. ROIs can also be filtered by a minimal "object" score or a minimal size. In the TensorFlow Model Garden implementation, these thresholds are available even if they are set to zero by default. Bad ROIs

can still be classified as "background" and rejected by the next stage, so letting them through at the RPN level is not a big problem.

One important practical consideration is that the RPN can be simple and fast if needed (see the example in Figure 4-23). It can use the output of the backbone directly, instead of using an FPN, and its classification and detection heads can use fewer convolutional layers. The goal is only to compute approximate ROIs around likely objects. They will be refined and classified in the next step.

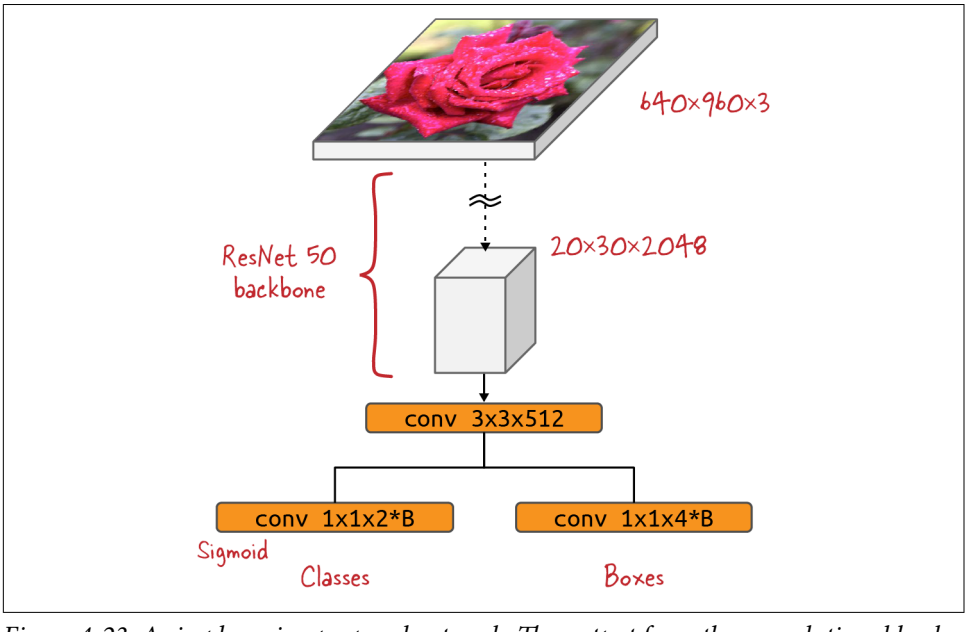


Figure 4-23. A simple region proposal network. The output from the convolutional backbone is fed through a two-class classification head (object or background) and a box regression head. B is the number of anchor boxes per location (typically three). An FPN can be used as well.

For example, the Mask R-CNN implementation in the TensorFlow Model Garden uses an FPN in its RPN but uses only three anchors per location, with aspect ratios of 0.5, 1.0, and 2.0, instead of the nine anchors per location used by RetinaNet.

R-CNN

We now have a set of proposed regions of interest. What next?

Conceptually, the R-CNN idea (Figure 4-24) is to crop the images along the ROIs and run the cropped images through the backbone again, this time with a full classification head attached to classify the objects (in our example, into "butterfly," "spider," etc.).

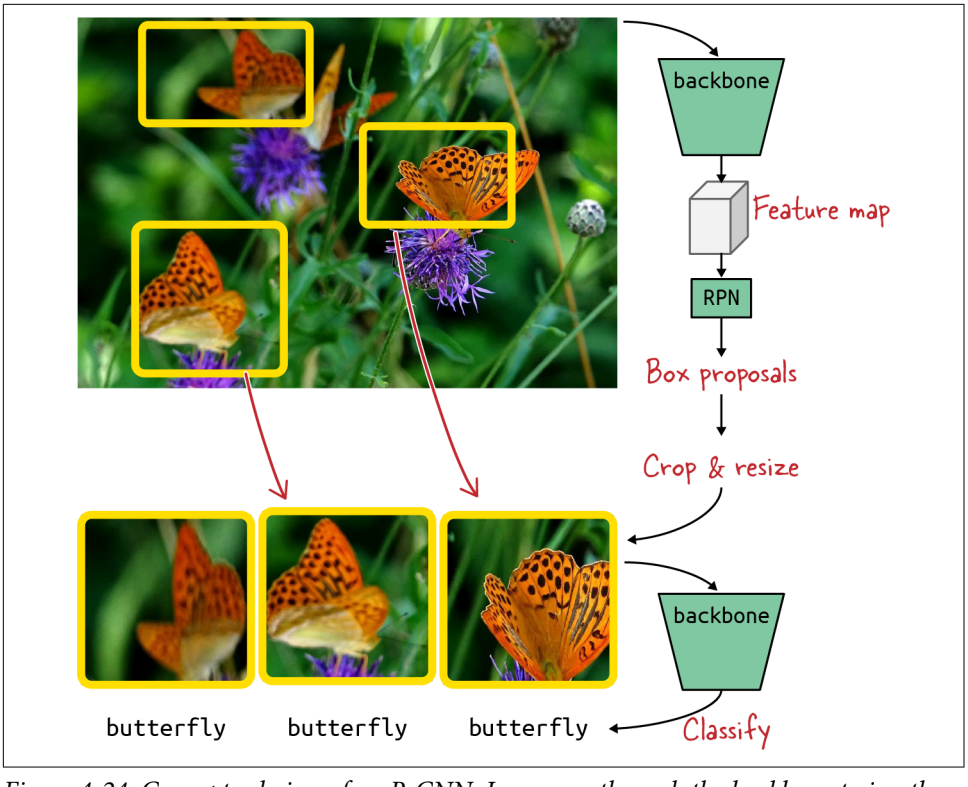


Figure 4-24. Conceptual view of an R-CNN. Images go through the backbone twice: the first time to generate regions of interest and the second time to classify the contents of these ROIs. Image from Arthropods dataset.

In practice, however, this is too slow. The RPN can generate somewhere in the region of 50 to 2,000 proposed ROIs, and running them all through the backbone again would be a lot of work. Instead of cropping the image, the smarter thing to do is to crop the feature map directly, then run prediction heads on the result, as depicted in Figure 4-25.

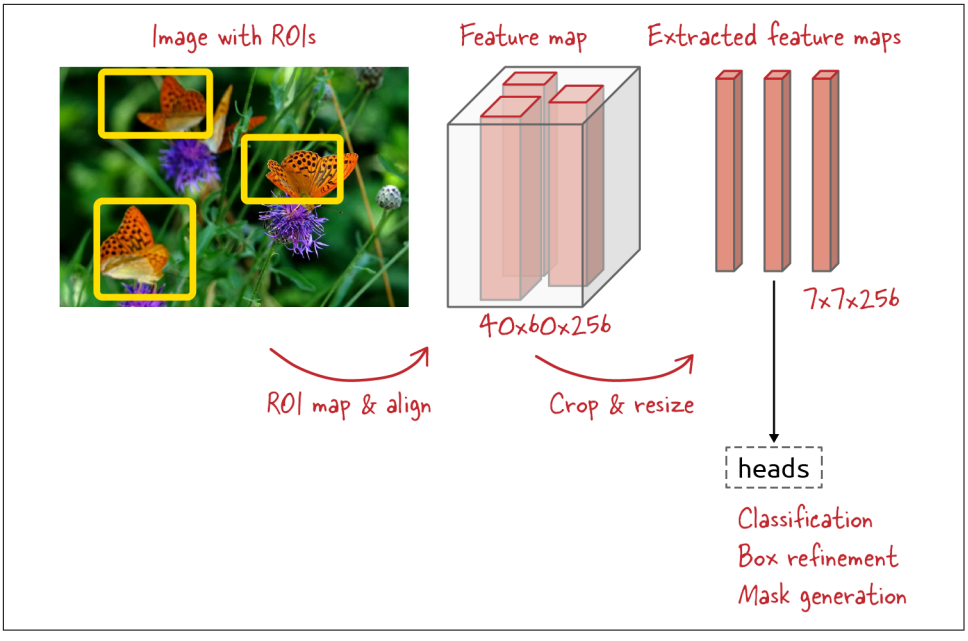


Figure 4-25. A faster R-CNN or Mask R-CNN design. As previously, the backbone generates a feature map and the RPN predicts regions of interest from it (only the result is shown). Then the ROIs are mapped back onto the feature map, and features are extracted and sent to the prediction heads for classification and more. Image from Arthropods dataset.

This is slightly more complex when an FPN is used. The feature extraction is still performed on a given feature map, but in a FPN there are several feature maps to choose from. A ROI therefore must first be assigned to the most relevant FPN level. The assignment is usually done using this formula:

$$n = floor\left(n_0 + log_2(\sqrt{wh}/224)\right)$$

where w and h are the width and height of the ROI, and n_0 is the FPN level where typical anchor box sizes are closest to 224. Here, *floor* stands for rounding down to the most negative number. For example, here are the typical Mask R-CNN settings:

- Five FPN levels, P_2, P_3, P_4, P_5, and P_6 (reminder: level P_n represents a feature map 2^n times smaller in width and height than the input image)
- Anchor box sizes of 32x32, 64x64, 128x128, 256x256, and 512x512 on their respective levels (same as in RetinaNet)
- $n_0 = 4$

With these settings, we can verify that (for example) an ROI of 80x160 pixels would get assigned to level P_3 and an ROI of 200x300 to level P_4, which makes sense.

ROI resampling (ROI alignment)

Special care is needed when extracting the feature maps corresponding to the ROIs. The feature maps must be extracted and resampled correctly. The Mask R-CNN paper's authors discovered that any rounding error made during this process adversely affects detection performance. They called their precise resampling method *ROI alignment*.

For example, let's take an ROI of 200x300 pixels. It would be assigned to FPN level P_4, where its size relative to the P_4 feature map becomes (200 / 2^4, 300 / 2^4) = (12.5, 18.75). These coordinates should not be rounded. The same applies to its position.

The features contained in this 12.5x18.75 region of the P_4 feature map must then be sampled and aggregated (using either max pooling or average pooling) into a new feature map, typically of size 7x7. This is a well-known mathematical operation called *bilinear interpolation,* and we won't dwell on it here. The important point to remember is that cutting corners here degrades performance.

Class and bounding box predictions

The rest of the model is pretty standard. The extracted features go through multiple prediction heads in parallel—in this case:

- A classification head to assign a class to each object suggested by the RPN, or classify it as background
- A box refinement head that further adjusts the bounding box

To compute detection and classification losses, the same target box assignment algorithm is used as in RetinaNet, described in the previous section. The box loss is also the same (Huber loss). The classification head uses a softmax activation with a special class added for "background." In RetinaNet it was a series of binary classifications. Both work, and this implementation detail is not important. The total training loss is the sum of the final box and classification losses as well as the box and classification losses from the RPN.

The exact design of the class and detection heads is given later, in Figure 4-30. They are also very similar to what was used in RetinaNet: a straight sequence of layers, shared between all levels of the FPN.

Mask R-CNN adds a third prediction head for classifying individual pixels of objects. The result is a pixel mask depicting the silhouette of the object (see Figure 4-19). It can be used if the training dataset contains corresponding target masks. Before we explain how it works, however, we need to introduce a new kind of convolution, one

capable of creating pictures rather than filtering and distilling them: transposed convolutions.

Transposed convolutions

Transposed convolutions, sometimes also called *deconvolutions*, perform a learnable upsampling operation. Regular upsampling algorithms like nearest neighbor upsampling or bilinear interpolation are fixed operations. Transposed convolutions, on the other hand, involve learnable weights.

The name "transposed convolution" comes from the fact that in the matrix representation of a convolutional layer, which we are not covering in this book, the transposed convolution is performed using the same convolutional matrix as an ordinary convolution, but transposed.

The transposed convolution pictured in Figure 4-26 has a single input and a single output channel. The best way to understand what it does is to imagine that it is painting with a brush on an output canvas. The brush is a 3x3 filter. Every value of the input image is projected through the filter on the output. Mathematically, every element of the 3x3 filter is multiplied by the input value and the result is added to whatever is already on the output canvas. The operation is then repeated at the next position: in the input we move by 1, and in the output we move with a configurable stride (2 in this example). Any stride larger than 1 results in an upsampling operation. The most frequent settings are stride 2 with a 2x2 filter or stride 3 with a 3x3 filter.

If the input is a feature map with multiple channels, the same operation is applied to each channel independently, with a new filter each time; then all the outputs are added element by element, resulting in a single output channel.

It is of course possible to repeat this operation multiple times on the same feature map, with a new set of filters each time, which results in a feature map with multiple channels.

In the end, for a multichannel input and a multichannel output, the weights matrix of a transposed convolution will have the shape shown in Figure 4-27. This is, by the way, the same shape as a regular convolutional layer.

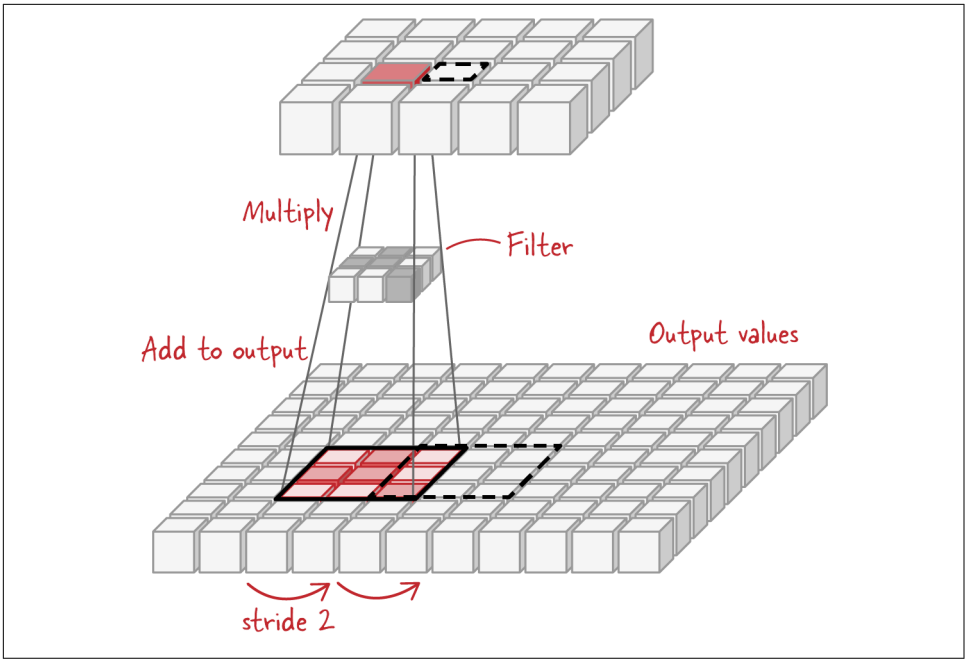


Figure 4-26. Transposed convolution. Each pixel of the original image (top) multiplies a 3x3 filter, and the result is added to the output. In a transposed convolution of stride 2, the output window moves by a step of 2 for every input pixel, creating a larger image (shifted output window pictured with a dashed outline).

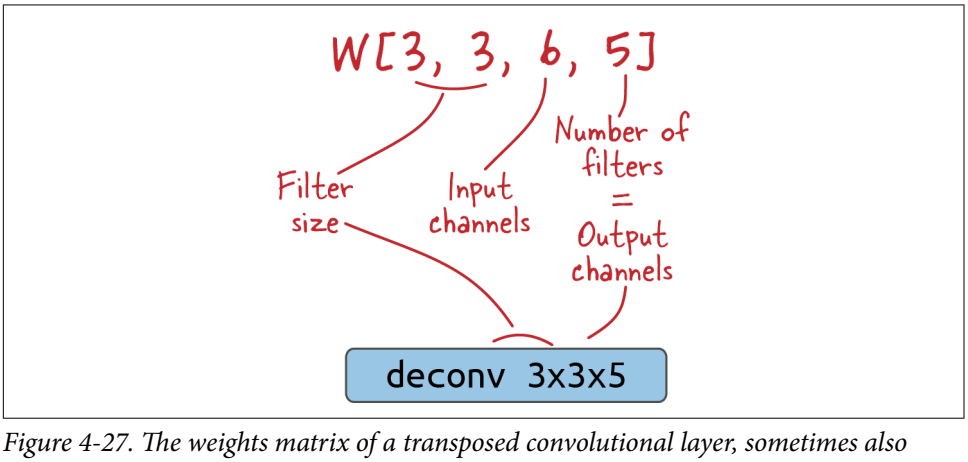


Figure 4-27. The weights matrix of a transposed convolutional layer, sometimes also called a "deconvolution." At the bottom is the schematic notation of deconvolutional layers that will be used for the models in this chapter.

Up-Convolution

Transposed convolutions are widely used in neural networks that generate images: autoencoders, generative adversarial networks (GANs), and so on. However, they have also been criticized for introducing "checkerboard" artifacts into the generated images (Odena et al., 2016), especially when their stride and filter size are not multiples of each other (Figure 4-28).

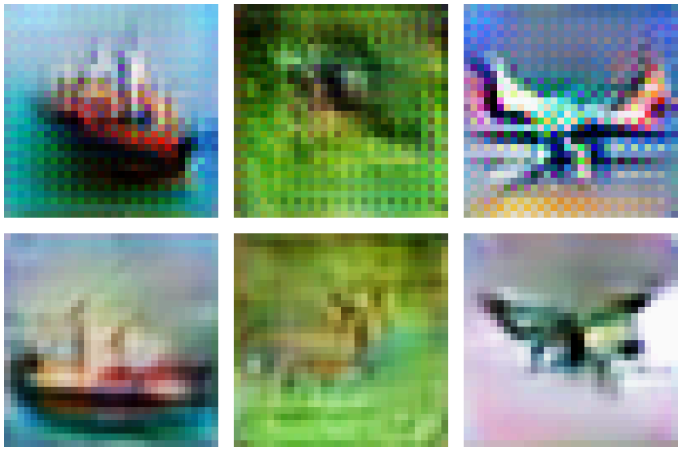


Figure 4-28. Transposed convolutions versus up-convolutions when used in a GAN. When transposed convolutions are used, an unwanted checkerboard pattern can appear (top row). This does not happen with up-convolutions. Image from Odena et al., 2016.

Odena et al. suggest using a simple nearest neighbour resampling followed by a regular convolution, a combination called "up-convolution" (Figure 4-29), instead of transposed convolutions. Interestingly, as you may recall from "Feature pyramid networks" on page 139, this is exactly how upsampling was handled there.

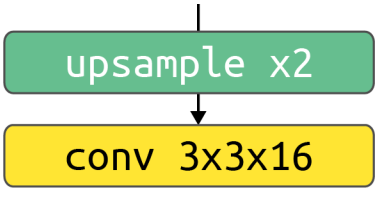


Figure 4-29. This is also a learnable upsampling operation. An "up-convolution" is a simple nearest neighbor upsampling operation followed by an ordinary convolutional layer.

Instance segmentation

Let's get back to Mask R-CNN, and its third prediction head that classifies individual pixels of objects. The output is a pixel mask outlining the silhouette of the object (see Figure 4-22).

Mask R-CNN and other RPNs work on a single ROI at a time, with a fairly high probability that this ROI is actually interesting, so they can do more work per ROI and with a higher precision. Instance segmentation is one such task.

The instance segmentation head uses transposed convolution layers to upsample the feature map into a black-and-white image that is trained to match the silhouette of the detected object.

Figure 4-30 shows the complete Mask R-CNN architecture.

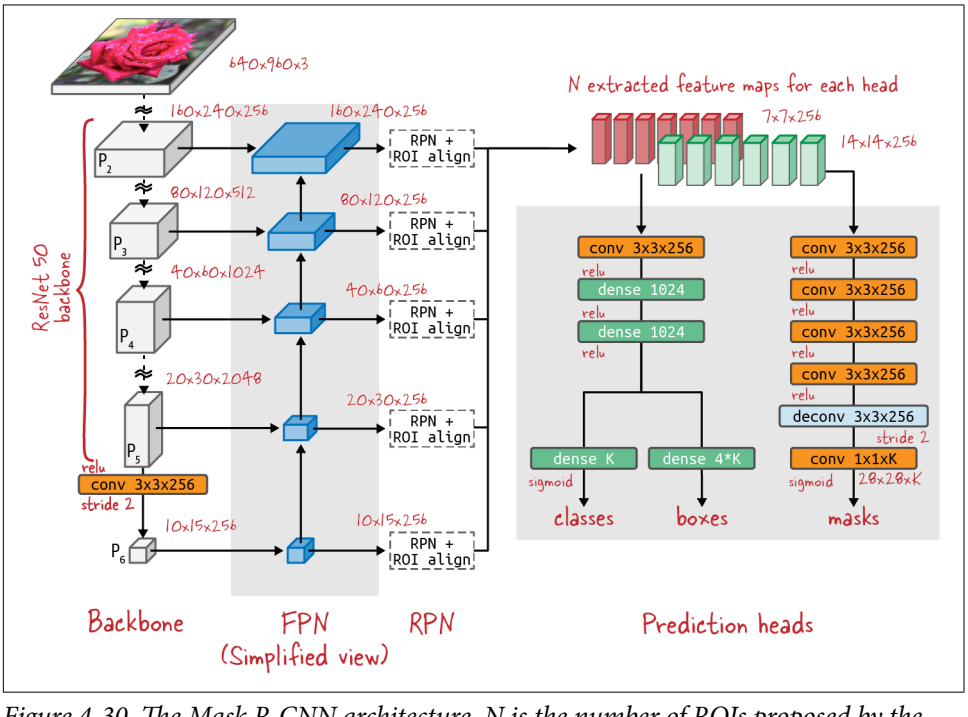


Figure 4-30. The Mask R-CNN architecture. N is the number of ROIs proposed by the RPN, and K is the number of classes; "deconv" denotes a transposed convolutional layer, which upsamples the feature maps to predict an object mask.

Notice that the mask head produces one mask per class. This seems to be redundant since there is a separate classification head. Why predict *K* masks for one object? In reality, this design choice increases the segmentation accuracy because it allows the segmentation head to learn class-specific hints about objects.

Another implementation detail is that the resampling and alignment of the feature maps to the ROIs is actually performed twice: once with a 7x7x256 output for the classification and detection head, and again with different settings (resampling to 14x14x256) specifically for the mask head to give it more detail to work with.

The segmentation loss is a simple pixel-by-pixel binary cross-entropy loss, applied once the predicted mask has been rescaled and upsampled to the same coordinates as the ground truth mask. Note that only the mask predicted for the predicted class is taken into account in the loss calculation. Other masks computed for the wrong classes are ignored.

We now have a complete picture of how Mask R-CNN works. One thing to notice is that with all the improvements added to the R-CNN family of detectors, Mask R-CNN is now a "two-pass" detector in name only. The input image effectively goes through the system only once. The architecture is still slower than RetinaNet but achieves a slightly higher detection accuracy and adds instance segmentation.

An extension of RetinaNet with an added mask head exists (RetinaMask), but it does not outperform Mask R-CNN. Interestingly, the paper notes that adding the mask head and associated loss actually improves the accuracy of bounding box detections (the other head). A similar effect might explain some of the improved accuracy of Mask R-CNN too.

One limitation of the Mask R-CNN approach is that the predicted object masks are fairly low resolution: 28x28 pixels. The similar but not exactly equivalent problem of semantic segmentation has been solved with high-resolution approaches. We'll explore this in the next section.

U-Net and Semantic Segmentation

In semantic segmentation, the goal is to classify every pixel of the image into global classes like "road," "sky," "vegetation," or "people" (see Figure 4-31). Individual instances of objects, like individual people, are not separated. All "people" pixels across the entire image are part of the same "segment."

Figure 4-31. In semantic image segmentation, every pixel in the image is assigned a category (like "road," "sky," "vegetation," or "building"). Notice that "people," for example, is a single class across the whole image. Objects are not individualized. Image from Cityscapes.

For semantic image segmentation, a simple and quite often sufficient approach is called U-Net. The U-Net is a convolutional network architecture that was designed for biomedical image segmentation (see Figure 4-32) and won a cell tracking competition in 2015.

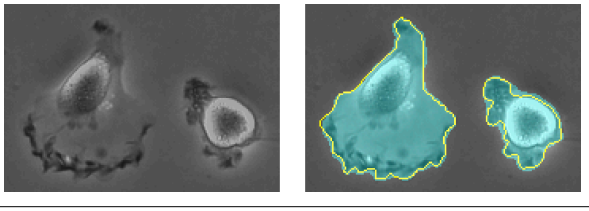


Figure 4-32. The U-Net architecture was designed to segment biomedical images such as these microscopy cell images. Images from Ronneberger et al., 2015.

The U-Net architecture is represented in Figure 4-33. A U-Net consists of an encoder which downsamples an image to an encoding (the lefthand side of architecture), and a mirrored decoder which upsamples the encoding back to the desired mask (the righthand side of the architecture). The decoder blocks have a number of skip connections (depicted by the horizontal arrows in the center) that directly connect from the encoder blocks. These skip connections copy features at a specific resolution and concatenate them channel-wise with specific feature maps in the decoder. This brings information at various levels of semantic granularity from the encoder directly into the decoder. (Note: cropping may be necessary on the skip connections because of slight size misalignments of the feature maps in corresponding levels of the encoder and decoder. Indeed, U-Net uses all convolutions without padding, which means that border pixels are lost at each layer. This design choice is not fundamental though, and padding can be used as well.)

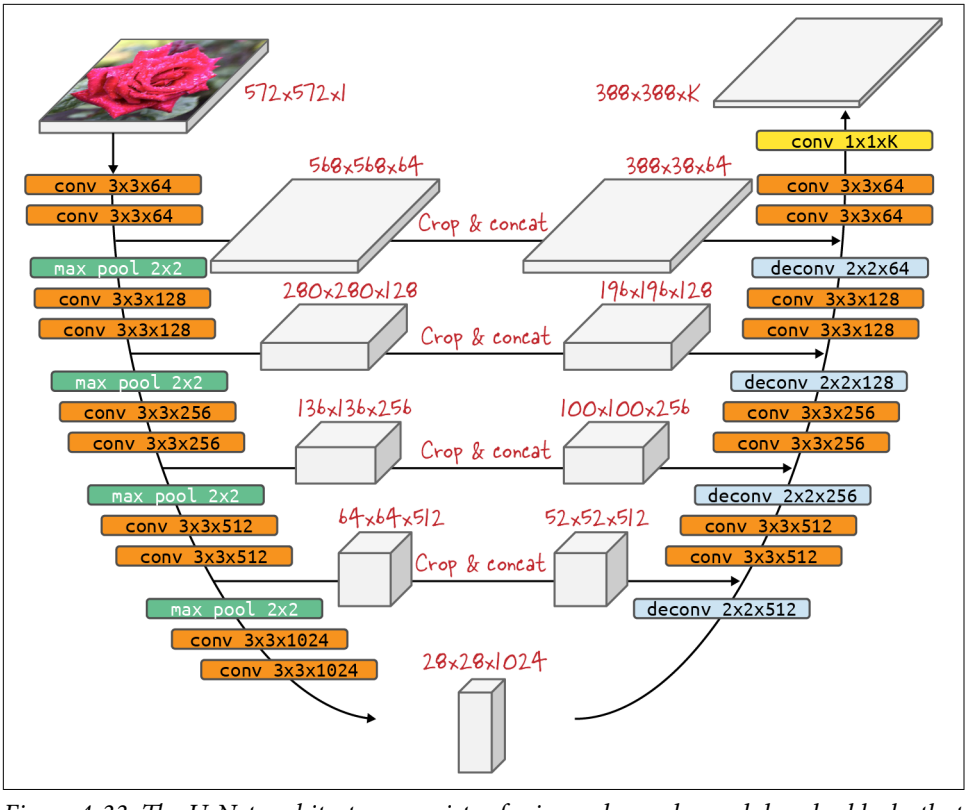


Figure 4-33. The U-Net architecture consists of mirrored encoder and decoder blocks that take on a U shape when depicted as shown here. Skip connections concatenate feature maps along the depth axis (channels). K is the target number of classes.

Images and labels

To illustrate U-Net image segmentation we'll use the Oxford Pets dataset, where each of the input images contains a label mask as shown in Figure 4-34. The label is an image in which pixels are assigned one of three integer values depending on whether they are background, the object outline, or the object interior.

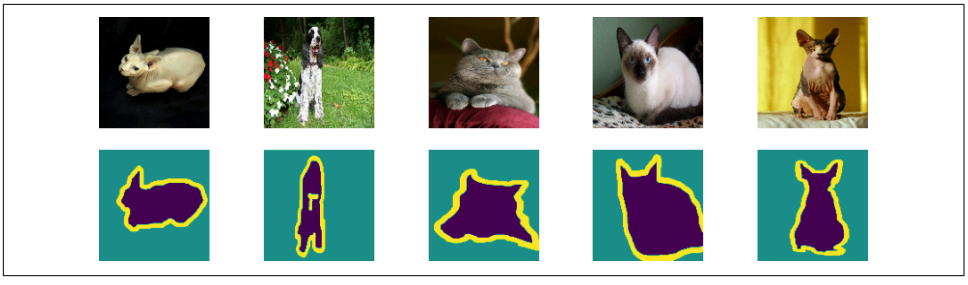


Figure 4-34. Training images (top row) and labels (bottom row) from the Oxford Pets dataset.

We'll treat these three pixel values as the index of class labels and train the network to carry out multiclass classification:

```
model = ...
model.compile(optimizer='adam',
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy'])
model.fit(...)
```

The complete code is available in *04b_unet_segmentation.ipynb* on GitHub.

Architecture

Training a U-Net architecture from scratch requires a lot of trainable parameters. As discussed in "Other considerations" on page 154 it's difficult to label datasets for tasks such as object detection and segmentation. Therefore, to use the labeled data efficiently, it is better to use a pretrained backbone and employ transfer learning for the encoder block. As in Chapter 3, we can use a pretrained MobileNetV2 to create the encoding:

```
base_model = tf.keras.applications.MobileNetV2(
    input_shape=[128, 128, 3], include_top=False)
```

The decoder side will consist of upsampling layers to get back to the desired mask shape. The decoder also needs feature maps from specific layers of the encoder (skip connections). The layers of the MobileNetV2 model that we need can be obtained by name as follows:

```
layer_names = [
    'block_1_expand_relu',   # 64x64
    'block_3_expand_relu',   # 32x32
    'block_6_expand_relu',   # 16x16
    'block_13_expand_relu',  # 8x8
    'block_16_project',      # 4x4
]
base_model_outputs = [base_model.get_layer(name).output for name in layer_names]
```

The "down stack" or lefthand side of the U-Net architecture then consists of the image as input, and these layers as outputs. We are carrying out transfer learning, so the entire lefthand side does not need weight adjustments:

```
down_stack = tf.keras.Model(inputs=base_model.input,
                            outputs=base_model_outputs,
                            name='pretrained_mobilenet')
down_stack.trainable = False
```

Upsampling in Keras can be accomplished using a `Conv2DTranspose` layer. We also add batch normalization and nonlinearity to each step of the upsampling:

```
def upsample(filters, size, name):
  return tf.keras.Sequential([
     tf.keras.layers.Conv2DTranspose(filters, size,
                                     strides=2, padding='same'),
     tf.keras.layers.BatchNormalization(),
     tf.keras.layers.ReLU()
  ], name=name)

up_stack = [
    upsample(512, 3, 'upsample_4x4_to_8x8'),
    upsample(256, 3, 'upsample_8x8_to_16x16'),
    upsample(128, 3, 'upsample_16x16_to_32x32'),
    upsample(64, 3,  'upsample_32x32_to_64x64')
]
```

Each stage of the decoder up stack is concatenated with the corresponding layer of the encoder down stack:

```
for up, skip in zip(up_stack, skips):
    x = up(x)
    concat = tf.keras.layers.Concatenate()
    x = concat([x, skip])
```

Training

We can display the predictions on a few selected images using a Keras callback:

```
class DisplayCallback(tf.keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs=None):
      show_predictions(train_dataset, 1)

model.fit(train_dataset, ...,
          callbacks=[DisplayCallback()])
```

The result of doing so on the Oxford Pets dataset is shown in Figure 4-35. Note that the model starts out with garbage (top row), as one would expect, but then learns which pixels correspond to the animal and which pixels correspond to the background.

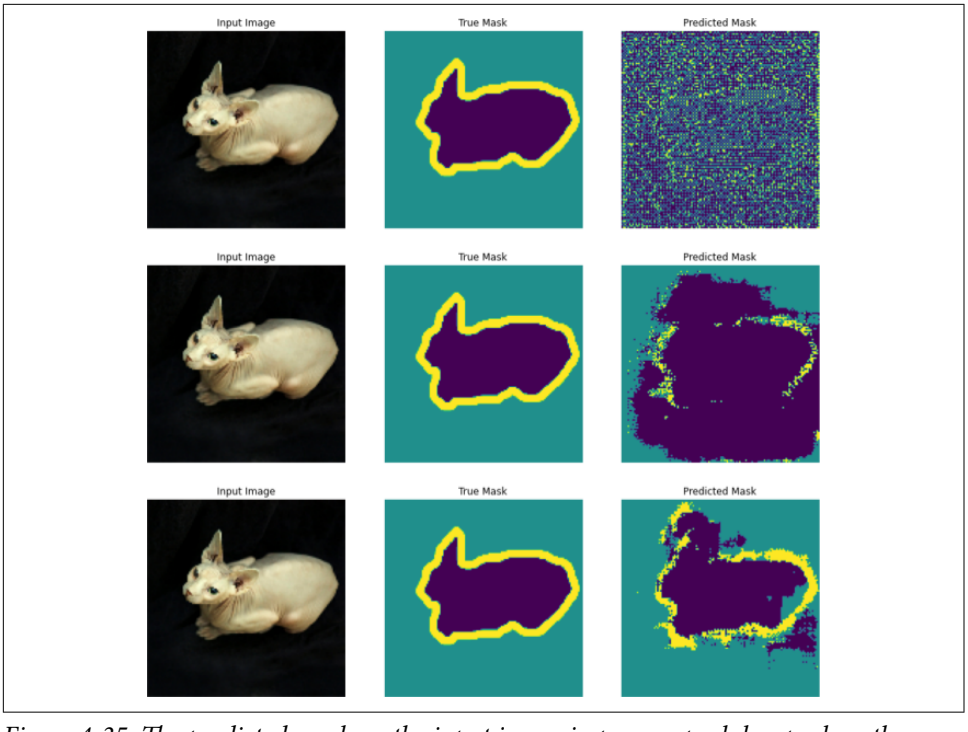


Figure 4-35. The predicted mask on the input image improves epoch by epoch as the model is trained.

However, because the model is trained to predict each pixel as background, outline, or interior independently of the other pixels, we see artifacts such as unclosed regions and disconnected pixels. The model doesn't realize that the region corresponding to the cat should be closed. That is why this approach is mostly used on images where the segments to be detected do not need to be contiguous, like the example in Figure 4-31, where the "road," "sky," and "vegetation" segments often have discontinuities.

An example of an application is in self-driving algorithms, to detect the road. Another is in satellite imagery, where a U-Net architecture was used to solve the hard problem of distinguishing clouds from snow; both are white, but snow coverage is useful ground-level information, whereas cloud obstruction means that the image needs to be retaken.

Current Research Directions

For object detection, a recent landmark is EfficientDet, which builds an object detection architecture on top of the EfficientNet backbone. Apart from the backbone, it brings a new class of multilayer FPNs called the bi-directional feature pyramid network (BiFPN).

For semantic segmentation, U-Net is the simplest but not the state-of-the-art architecture. You can read about more complex approaches in the papers on DeepLabv3 and the pyramid scene parsing network (PSPNet). DeepLabv3 is currently considered state of the art for image segmentation and is the architecture implemented in the TensorFlow Model Garden.

Kirillov et al. recently opened up a new field in object detection with their proposed panoptic segmentation task, which combines instance segmentation and full-scene semantic segmentation into a single model. Objects are classified in two ways: either countable objects like "persons" or "cars" that are detected individually, or uncountable types of objects like "the road" or "the sky" which are segmented globally. The current top approaches in this domain are Panoptic FPN and Panoptic DeepLab, both implemented as part of the Detectron2 platform.

Summary

In this chapter, we looked at object detection and image segmentation methods. We started with YOLO, considering its limitations, and then discussed RetinaNet, which innovates over YOLO in terms of both the architecture and the losses used. We also discussed Mask R-CNN to carry out instance segmentation and U-Net to carry out semantic segmentation.

In the next chapters, we will delve more deeply into different parts of the computer vision pipeline using the simple transfer learning image classification architecture from Chapter 3 as our core model. The pipeline steps remain the same regardless of the backbone architecture or the problem being solved.

CHAPTER 5

Creating Vision Datasets

To carry out machine learning on images, we need images. Of the use cases we looked at in Chapter 4, the vast majority were for supervised machine learning. For such models, we also need the correct answer, or *label*, to train the ML model. If you are going to train an unsupervised ML model or a self-supervised model like a GAN or autoencoder, you can leave out the labels. In this chapter, we will look at how to create a machine learning dataset consisting of images and labels.

The code for this chapter is in the *05_create_dataset* folder of the book's GitHub repository. We will provide file names for code samples and notebooks where applicable.

Collecting Images

In most ML projects, the first stage is to collect the data. The data collection might be done in any number of ways: by mounting a camera at a traffic intersection, connecting to a digital catalog to obtain photographs of auto parts, purchasing an archive of satellite imagery, etc. It can be a logistical activity (mounting traffic cameras), a technical activity (building a software connector to the catalog database), or a commercial one (purchasing an image archive).

Metadata as Input

Because the success of an ML project is highly dependent on the quality and quantity of the data, we might need to collect not just the image data but also a variety of metadata about the context in which these images were acquired (for example, the weather at the time that a photograph of the traffic intersection was taken, and the status of all the traffic lights at that intersection). Although we will focus only on images in this book, remember that you can dramatically improve the accuracy of your ML model by feeding such metadata into one of the late-stage dense layers of your image model as contextual features. In essence, this concatenates the metadata with the image embeddings and trains a dense neural network in conjunction with the image model. For more details, see the discussion of the Multimodal Input design pattern in Chapter 2 of the book *Machine Learning Design Patterns* by Valliappa Lakshmanan, Sara Robinson, and Michael Munn (O'Reilly).

Photographs

Photographs are one of the most common sources of image data. These can include photographs taken from social media and other sources, and photographs taken under controlled conditions by permanently mounted cameras.

One of the first choices we need to make when collecting images is the placement of the camera and the size and resolution of the image. Obviously, the image has to frame whatever it is that we are interested in—for example, a camera mounted to photograph a traffic intersection would need to have an unobstructed view of the entire intersection.

Intuitively, it might seem that we will get the highest-accuracy models by training them on the highest-resolution images, and so we should endeavor to collect data at the highest resolution we can. However, high image resolutions come with several drawbacks:

- Larger images will require larger models—the number of weights in every layer of a convolutional model scales proportionally to the size of the input image. We'll need four times the number of parameters to train a model on 256x256 images than on 128x128 images, so training will take longer and will require more computational capacity and additional memory.
- The machines that we train ML models on have limited memory (RAM), so the larger the image size, the fewer images we can have in a batch. In general, larger batch sizes lead to smoother training curves. So, large images may be counterproductive in terms of accuracy.

- Higher-resolution images, especially those taken in outdoor and low-light environments, may have more noise. Smoothing the images down to a lower resolution might lead to faster training and higher accuracy.
- Collecting and saving a high-resolution image takes longer than collecting and saving a lower-resolution one. Therefore, to capture high-speed action, it might be necessary to use a lower resolution.
- Higher-resolution images take longer to transmit. So, if you are collecting images on the edge[1] and sending them to the cloud for inference, you can do faster inference by using smaller, lower-resolution images.

The recommendation, therefore, is to use the highest resolution that is warranted by the noise characteristics of your images and that your machine learning infrastructure budget can handle. Do not lower the resolution so much that the objects of interest cannot be resolved.

In general, it is worth using the highest-quality camera (in terms of lens, sensitivity, and so on) that your budget will allow—many computer vision problems are simplified if the images used during prediction will always be in focus, if the white balance will be consistent, and if the effect of noise on the images is minimal. Some of these problems can be rectified using image preprocessing (image preprocessing techniques will be covered in Chapter 6). Still, it is better to have images without these issues than to collect data and have to correct them after the fact.

Cameras can typically save photographs in a compressed (e.g., JPEG) or uncompressed (e.g., RAW) format. When saving JPEG photographs, we can often choose the quality. Lower-quality and lower-resolution JPEG files compress better, and so incur lower storage costs. As described previously, lower-resolution images will also reduce compute costs. Because storage is inexpensive relative to compute, our recommendation is to choose a high-quality threshold for the JPEGs (95%+) and store them at a lower resolution.

The lowest resolution you can use depends on the problem. If you are trying to classify landscape photographs to determine whether they are of water or land, you might be able to get away with 12x16 images. If your aim is to identify the type of trees in those landscape photographs, you might need the pixels to be small enough to clearly pick up on the shapes of leaves, so you might need 768x1024 images.

1 This is common in Internet of Things (IoT) applications; see the Wikipedia entry on "Fog computing".

Use uncompressed images only if your images consist of human-generated content such as CAD drawings where the fuzzy edges of the JPEG compression might cause issues with recognition.

Imaging

Many instruments (X-rays, MRIs, spectroscopes, radars, lidars, and so on) create 2D or 3D images of a space. X-rays are projections of a 3D object and may be treated as grayscale images (see Figure 5-1). While typical photographs contain three (red, green, blue) channels, these images have only one channel.

Figure 5-1. A chest X-ray can be treated as a grayscale image. Image courtesy of Google AI Blog.

If the instrument measures multiple quantities, we can treat the reflectance at different wavelengths, Doppler velocity, and any other measured quantities as separate channels of the image. In tomography, the projections are of thin 3D slices, thus creating multiple cross-sectional images; these cross-sections may be treated as channels of a single image.

There are some special considerations associated with imagery data depending on the sensor geometry.

Polar grids

Radar and ultrasound are carried out in a polar coordinate system (see Figure 5-2). You can either treat the polar 2D data itself as the input image, or transform it into a Cartesian coordinate system before using it as an input to machine learning models. There are trade-offs in either approach: in the polar coordinate system there is no interpolation or repeated pixels but the size of a pixel varies throughout the image, whereas in the Cartesian coordinate system the pixel size is consistent but much of the data is missing, interpolated, or aggregated when remapped. For example, in Figure 5-2, many of the pixels at the bottom left will be missing and will have to be

assigned some numerical value for ML purposes. Meanwhile, pixels at the bottom right will involve aggregation of many values from the pixel grid and pixels at the top will involve interpolation between the pixel values. The presence of all three situations in Cartesian images greatly complicates the learning task.

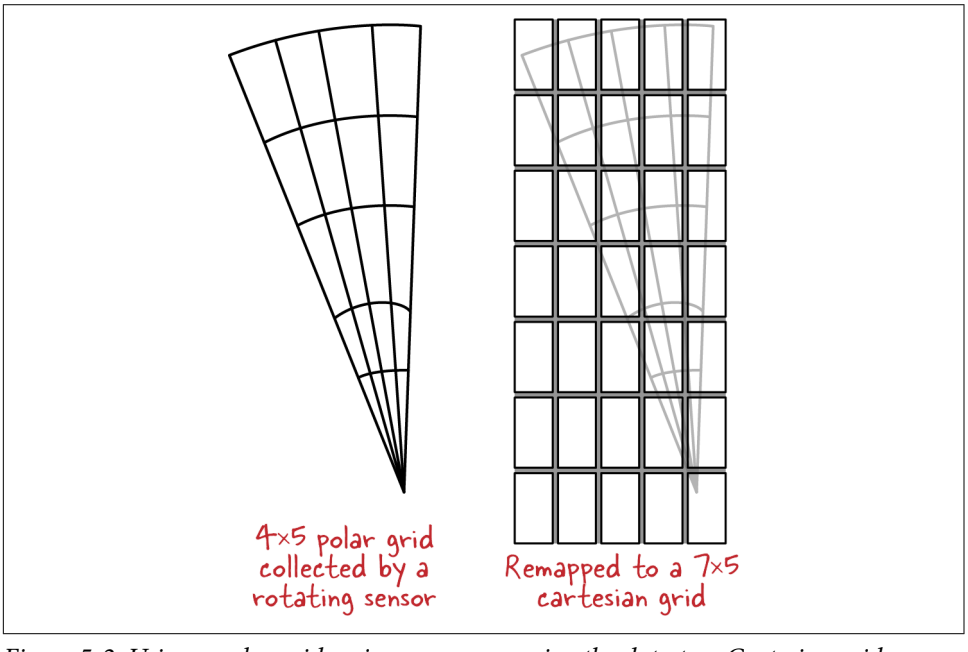


Figure 5-2. Using a polar grid as-is versus remapping the data to a Cartesian grid.

We recommend using the polar grid as the input image to ML models, and including the distance of each pixel from the center (or the size of the pixel) as an additional input to the ML model. Because every pixel has a different size, the easiest way to incorporate this information is to treat the size of the pixel as an additional channel. This way we can take advantage of all of the image data without losing information through coordinate transformation.

Satellite channels

When working with satellite images, it might be worth working in the original satellite view or a parallax-corrected grid rather than remapping the images to Earth coordinates. If using projected map data, try to carry out the machine learning in the original projection of the data. Treat images collected of the same location at approximately the same time, but at different wavelengths, as channels (see Figure 5-3). Note that pretrained models are usually trained on three-channel images (RGB), so transfer learning and fine-tuning will not work, but the underlying architectures can work with any number of channels if you are training from scratch.

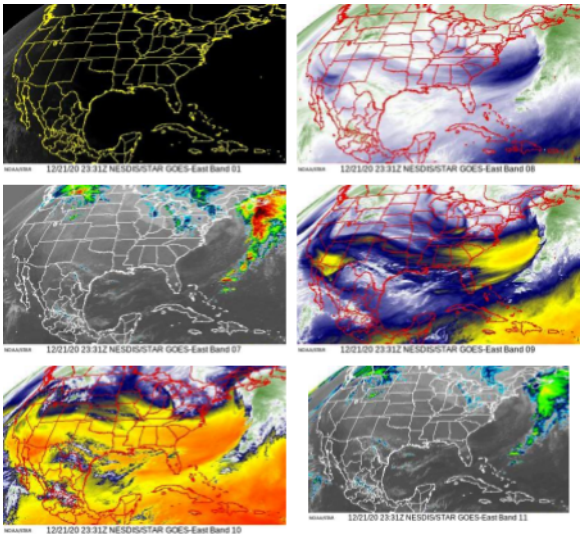


Figure 5-3. Images collected by instruments onboard the GOES-16 weather satellite at approximately the same time on December 21, 2020. Treat the original scalar values of these colorized images as six-channel images that are input to the model. Images courtesy of the US National Weather Service.

Geospatial layers

If you have multiple map layers (e.g., land ownership, topography, population density; see Figure 5-4) collected in different projections, you will have to remap them into the same projection, line up the pixels, and treat these different layers as channels of an image. In such situations, it might be useful to include the latitude of the pixel as an additional input channel to the model so that changes in pixel size can be accounted for.

Categorical layers (such as land cover type) may have to be one-hot encoded so that the land cover type becomes five channels if there are five possible land cover types.

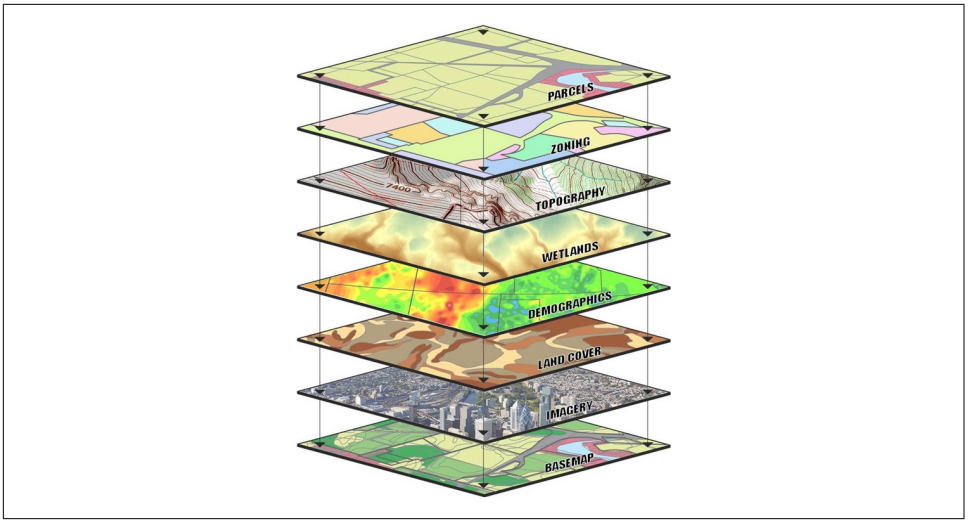


Figure 5-4. Geospatial layers can be treated as image channels. Image courtesy of USGS.

Proof of Concept

In many situations, you may not have the data on hand, and collecting it for a proof of concept would take too long. You may look into purchasing similar data to understand the feasibility of a project before investing in routine data collection. When purchasing images, keep in mind that you want to acquire images that are similar in quality, resolution, etc. to the images that you will ultimately be able to use in the actual project.

For example, many of the machine learning algorithms for the US GOES-16 satellite had to be developed before the satellite was launched. Naturally, there was no data available! In order to decide on the list of ML models that would be built on the GOES-16 data, similar-quality data already being collected by the European SEVIRI satellite was used to carry out proof-of-concept tests.

Another way to carry out a proof of concept is to simulate images. We will see an example of this in Chapter 11, where the ability to count tomatoes on a vine is illustrated through simulated images. When simulating images, it can be helpful to modify existing images rather than creating them from scratch. For example, the simulated tomato vine images might have been easier to generate if photographs of green vines, to which red tomatoes of different sizes could be added, had been readily available.

Do not train a model on perfect data and then try to apply it to imperfect images. For example, if you need a model to be able to identify flowers from photographs that hikers take on trails, you should not train the model on photographs taken by professional photographers that were subsequently retouched.

Data Types

So far, we have processed only photographs. As discussed in the previous section, there are other types of images, such as geospatial layers, MRI scans, or spectrograms of sound, to which machine learning can be applied. Mathematically, all that the ML techniques require is a 4D tensor (batch x height x width x channels) as input. As long as our data can be put into this form, computer vision methods may be applied.

Of course, you have to keep in mind the underlying concepts that make certain techniques work well. For example, you may not find success in applying convolutional filters to the problem of finding defective pixels on computer monitors because convolutional filters work well only when there is spatial correlation between adjacent pixels.

Channels

A typical photograph is stored as a 24-bit RGB image with three channels (red, green, and blue), each of which is represented by an 8-bit number in the range 0–255. Some computer-generated images also have a fourth *alpha* channel, which captures the transparency of the pixel. The alpha channel is useful primarily to overlay or composite images together.

Scaling

Machine learning frameworks and pretrained models often expect that the pixel values are scaled from [0,255] to [0,1]. ML models typically ignore the alpha channel. In TensorFlow, this is done using:

```
# Read compressed data from file into a string.
img = tf.io.read_file(filename)
# Convert the compressed string to a 3D uint8 tensor.
img = tf.image.decode_jpeg(img, channels=3)
# Convert to floats in the [0,1] range.
img = tf.image.convert_image_dtype(img, tf.float32)
```

Channel order

The shape of a typical image input is [height, width, channels], where the number of channels is typically 3 for RGB images and 1 for grayscale. This is called a *channels-last* representation and is the default with TensorFlow. Earlier ML packages such as Theano and early versions of ML infrastructure such as Google's Tensor Processing Unit (TPU) v1.0 used a *channels-first* ordering. The channels-first order is more efficient in terms of computation because it reduces back-and-forth seeks within memory.[2] However, most image formats store the data pixel by pixel, so channels-last is the more natural data ingest and output format. The move from channels-first to channels-last is an example of ease of use being prioritized over efficiency as computational hardware becomes more powerful.

Because channel order can vary, Keras allows you to specify the order in the global *$HOME/.keras/keras.json* configuration file:

```
{
    "image_data_format": "channels_last",
    "backend": "tensorflow",
    "epsilon": 1e-07,
    "floatx": "float32"
}
```

The default is to use TensorFlow as the Keras backend, and therefore the image format defaults to `channels_last`. This is what we will do in this book. Because this is a global setting that will affect every model being run on the system, we strongly recommend that you don't fiddle with this file.

If you have an image that is channels-first and need to change it to channels-last, you can use `tf.einsum()`:

```
image = tf.einsum('chw->hwc', channels_first_image)
```

or simply do a transpose, providing the appropriate axes:

```
image = tf.transpose(channels_first_image, perm=(1, 2, 0))
```

2 See "Understanding Memory Formats" on the oneAPI Deep Neural Network Library.

tf.einsum

You can use `tf.einsum()` to apply element-wise computation in a simple way. This uses a shorthand form known as *Einstein summation*. For example, to transpose a matrix, you can simply do:

```
output = tf.einsum('ij->ji', m)
# output[j,i] = m[i,j]
```

A comma in Einstein summation is a matrix multiplication:

```
output = tf.einsum('i,j->ij', u, v)
# output[i,j] = u[i]*v[j]
```

The shorthand syntax also supports summation and broadcasting. See the TensorFlow documentation for details.

Grayscale

If you have a grayscale image, or a simple 2D array of numbers, you may have to expand the dimensions to change the shape from [height, width] to [height, width, 1]:

```
image = tf.expand_dims(arr2d, axis=-1)
```

By specifying `axis=-1`, we ask for the channel dimension to be appended to the existing shape and the new channel dimension to be set to 1.

Geospatial Data

Geospatial data can either be generated from map layers or as a result of remote sensing from drones, satellites, radars, and the like.

Raster data

Geospatial data that results from maps often has *raster bands* (2D arrays of pixel values) that can be treated as channels. For example, you may have several raster bands covering a land area: population density, land cover type, flooding propensity, and so on. In order to apply computer vision techniques to such raster data, simply read the individual bands and stack them together to form an image:

```
image = tf.stack([read_into_2darray(b) for b in raster_bands], axis=-1)
```

In addition to raster data, you might also have vector data such as the locations of roads, rivers, states, or cities. In that case, you have to rasterize the data before using it in image-based ML models. For example, you might draw the roads or rivers as a set of one-pixel-wide line segments (see the top panel of Figure 5-5). If the vector data consists of polygons, such as state boundaries, you would rasterize the data by filling in the pixels that fall within the boundary. If there are 15 states, then you will end up

with 15 raster images, with each image containing 1 in the pixels that are within the boundary of the corresponding state—this is the image equivalent of one-hot encoding categorical values (see the bottom panel of Figure 5-5). If the vector data consists of city boundaries, you'll have to decide whether to treat this as a Boolean value (the pixel value is 0 if rural, and 1 if it is a city) or a categorical variable (in which case, you'd generate *N* raster bands for the *N* cities in the dataset).

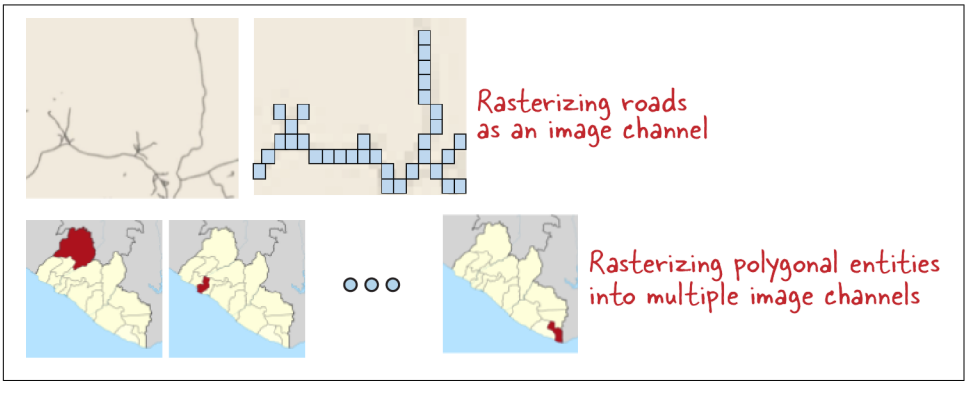


Figure 5-5. Rasterizing vector data. In the rasterized images, the 1s are highlighted. Map sources: OpenStreetMap (top) and Wikipedia (bottom).

The raster data will typically be in a geographic projection. Some projections (such as Lambert conformal) preserve areas, others (such as Mercator) preserve direction, and others (such as equidistant cylindrical) are chosen because they are simple to create. In our experience, any projection works fine for machine learning, but you should ensure that all the raster bands are in the same projection. It can also be helpful to add the latitude as an additional input channel if the size of a pixel will vary with latitude.

Remote sensing

Remotely sensed data is collected by an imaging instrument. If the instrument in question is a camera (as with a lot of drone images), the result will be an image with three channels. On the other hand, if there are multiple instruments on board the satellite capturing the images or if the instrument can operate at multiple frequencies, the result will be an image with a large number of channels.

Often, remote sensing images are colorized for visualization purposes. It is better to go back and get the raw numbers that are sensed by the instrument rather than using these colorized images.

Make sure that you read and normalize the images as we did for the photographs. For example, scale the values found in each image from 0 to 1. Sometimes the data will contain outliers. For example, bathymetric images may have outlier values due to ocean waves and tides. In such cases, it may be necessary to clip the data to a reasonable range before scaling it.

Remote sensing images will often contain missing data (such as the part of the image outside of the satellite horizon or areas of clutter in radar images). If it is possible to crop out the missing areas, do so. Impute missing values by interpolating over them if the missing areas are small. If the missing values consist of large areas, or occur in a significant fraction of the pixels, create a separate raster band that indicates whether the pixel is missing a true value or has been replaced by a sentinel value such as zero.

Both geospatial and remote sensing data require a significant amount of processing before they can be input into ML models. Because of this, it is worthwhile to have a scripted/automated data preparation step or pipeline that takes the raw images, processes them into raster bands, stacks them, and writes them out into an efficient format such as TensorFlow Records.

Audio and Video

Audio is a 1D signal whereas videos are 3D. It is better to use ML techniques that have been devised specifically for audio and video, but a simple first solution might involve applying image ML techniques to audio and video data. In this section, we'll discuss this approach. Audio and video ML frameworks are outside the scope of this book.

Spectrogram

To do machine learning on audio, it is necessary to split the audio into chunks and then apply ML to these time windows. The size of the time window depends on what's being detected—you need a few seconds to identify words, but a fraction of a second to identify instruments.

The result is a 1D signal, so it is possible to use `Conv1D` instead of `Conv2D` layers to process audio data. Technically speaking, this would be signal processing in the time space. However, the results tend to be better if audio signals are represented as spectrograms—a stacked view of the spectrum of frequencies in the audio signal as it varies over time. In a spectrogram, the x-axis of the image represents time and the y-axis represents the frequency. The pixel value represents the spectral density, which is the loudness of the audio signal at a specific frequency (see Figure 5-6). Typically, the spectral density is represented in decibels, so it is best to use the logarithm of the spectrogram as the image input.

To read and convert an audio signal into the log of the spectrogram, use the `scipy` package:

```
from scipy import signal
from scipy.io import wavfile
sample_rate, samples = wavfile.read(filename)
_, _, spectro = signal.spectrogram(samples, sample_rate)
img = np.log(spectro)
```

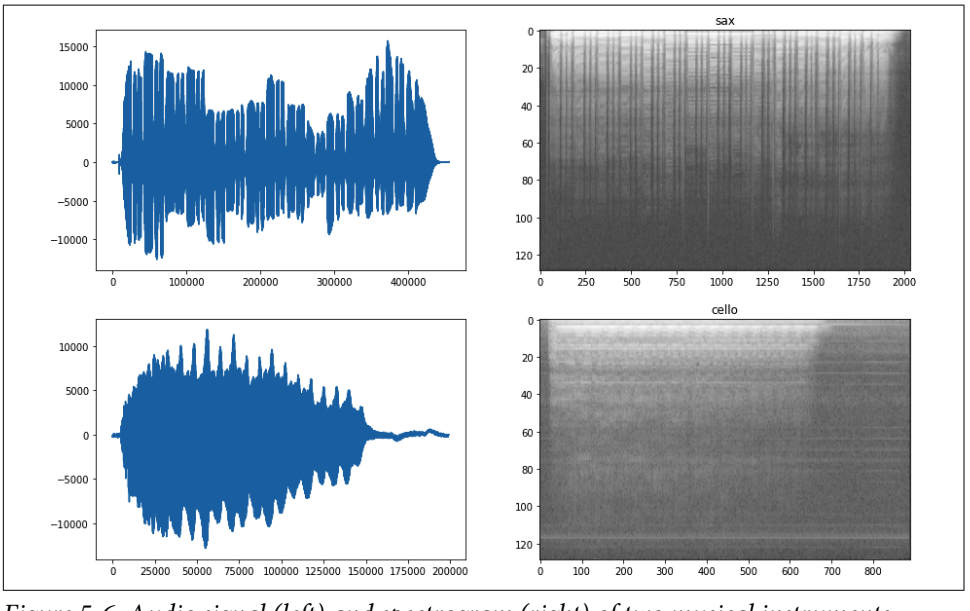


Figure 5-6. Audio signal (left) and spectrogram (right) of two musical instruments.

Natural Language Processing Using Computer Vision Techniques

In the spectrogram, we are computing the frequency characteristics of the signal at a point in time and then looking at the variation in this frequency to create a 2D image. This idea of grouping 1D objects into 2D in order to use computer vision techniques can also be applied to natural language processing problems!

For example, it is possible to take a document-understanding problem and treat it like a computer-vision problem. The idea is to use a pretrained embedding such as the Universal Sentence Encoder (USE) or BERT to convert sentences to embeddings (the full code is in *05_audio.ipynb* on GitHub):

```
paragraph = ...
embed = hub.load(
    https://tfhub.dev/google/universal-sentence-encoder/4")
embeddings = embed(paragraph.split('.'))
```

In this code snippet, we are splitting the paragraph into a list of sentences and applying the USE embedder to a set of sentences. Because `embeddings` is now a 2D tensor, it is possible to treat the paragraph/review/page/document (whatever your grouping unit is) as an image (see Figure 5-7). The image is 12x512 because there are 12 sentences in the paragraph and we used an embedding of size 512.

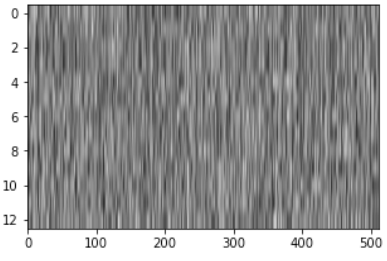


Figure 5-7. By stacking the text embeddings of sentences, we can treat a paragraph as an image. This is the representation of a paragraph from a Herman Hesse novel.

Frame by frame

Videos consist of *frames*, each of which is an image. The obvious approach to handling videos is to carry out image processing on the individual frames, and postprocess the results into an analysis of the entire video. We can use the OpenCV (`cv2`) package to read a video file in one of the standard formats and obtain a frame:

```
cap = cv2.VideoCapture(filename)
num_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
for i in range(num_frames):
    readok, frame = cap.read()
    if readok:
        img = tf.convert_to_tensor(frame)
```

For example, we might classify the image frames, and treat the result of the video classification problem as the set of all the categories found in all the frames. The problem is that such an approach loses sight of the fact that adjacent frames in a video are highly correlated, just as adjacent pixels in an image are highly correlated.

Conv3D

Instead of processing videos one frame at a time, we can compute rolling averages of video frames and then apply computer vision algorithms. This approach is particularly useful when the videos are grainy. Unlike the frame-by-frame approach, the rolling average takes advantage of frame correlation to denoise the image.

A more sophisticated approach is to use 3D convolution. We read video clips into a 5D tensor with the shape [batch, time, height, width, channels], breaking the movie into short clips if necessary:

```
def read_video(filename):
    cap = cv2.VideoCapture(filename)
    num_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    frames = []
    for i in range(num_frames):
        readok, frame = cap.read()
        if readok:
            frames.append(frame)
    return tf.expand_dims(tf.convert_to_tensor(frames), -1)
```

Then, we apply `Conv3D` instead of `Conv2D` in our image processing pipeline. This is similar to a rolling average where the weights of each time step are learned from the data, followed by a nonlinear activation function.

Another approach is to use recurrent neural networks (RNNs) and other sequence methods that are more suitable for time-series data. However, because RNNs of video sequences are quite hard to train, the 3D convolutional approach tends to be more practical. An alternative is to extract features from the time signal using convolution and then pass the results of the convolution filter to a less complex RNN.

Manual Labeling

In many ML projects, the first step at which the data science team gets involved is in labeling the image data. Even if the labeling will be automated, the first few images in a proof of concept are almost always hand-labeled. The form and organization will differ based on the problem type (image classification or object detection) and whether an image can have multiple labels or only one.

To hand-label images, a *rater* views the image, determines the label(s), and records the label(s). There are two typical approaches to doing this recording: using a folder structure and a metadata table.

In a folder organization, raters simply move images to different folders depending on what their label is. All flowers that are daisies are stored in a folder named *daisy*, for example. Raters can do this quickly because most operating systems provide previews of images and handy ways to select groups of images and move them into folders (see Figure 5-8).

The problem with the folder approach is that it leads to duplication if an image can have multiple labels—for example, if an image contains both roses and daisies.

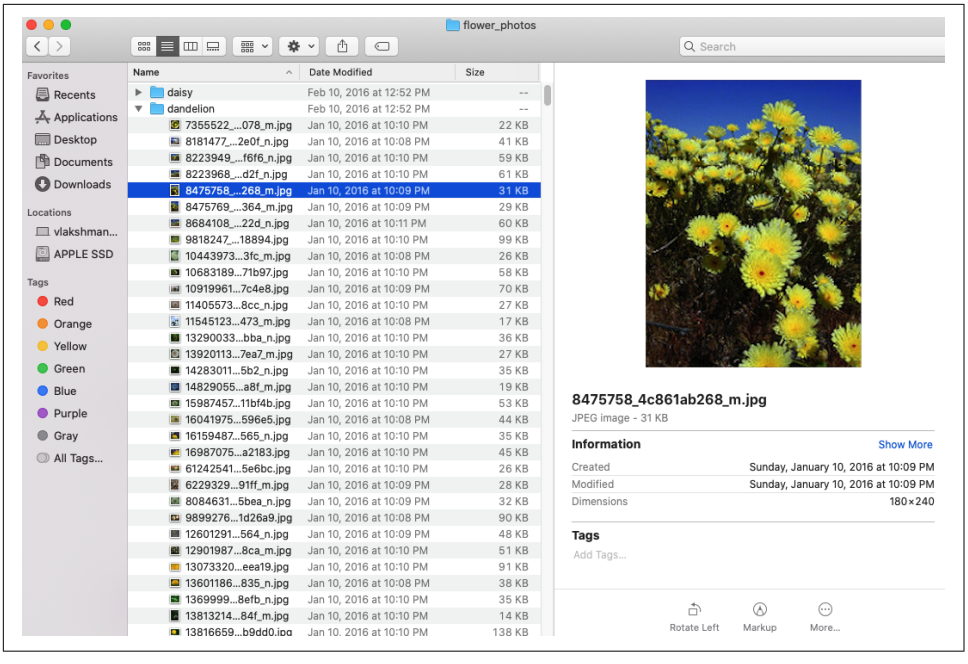


Figure 5-8. Preview images and quickly move them to the appropriate folder.

The alternative, and recommended, approach is to record the label(s) in a metadata table (such as in a spreadsheet or a CSV file) that has at least two columns—one column is the URL to the image file, and the other is the list of labels that are valid for the image:

```
$ gsutil cat gs://cloud-ml-data/img/flower_photos/all_data.csv | head -5
gs://cloud-ml-data/img/flower_photos/daisy/100080576_f52e8ee070_n.jpg,daisy
gs://cloud-ml-data/img/flower_photos/daisy/10140303196_b88d3d6cec.jpg,daisy
gs://cloud-ml-data/img/flower_photos/daisy/10172379554_b296050f82_n.jpg,daisy
gs://cloud-ml-data/img/flower_photos/daisy/10172567486_2748826a8b.jpg,daisy
gs://cloud-ml-data/img/flower_photos/daisy/10172636503_21bededa75_n.jpg,daisy
```

A good approach to marry the efficiency of the folder approach and the generalizability of the metadata table approach is to organize the images into folders and then use a script to crawl the images and create the metadata table.

Multilabel

If an image can be associated with multiple labels (for example, if an image can contain both daisies and sunflowers), one approach is to simply copy the image into both folders and have two separate lines:

```
gs://.../sunflower/100080576_f52e8ee070_n.jpg,sunflower
gs://.../daisy/100080576_f52e8ee070_n.jpg,daisy
```

However, having duplicates like this will make it more difficult to train a truly multi-label multiclass problem. A better approach is to make the labels column contain all the matching categories:

```
gs://.../multi/100080576_f52e8ee070_n.jpg,sunflower daisy
```

The ingest pipeline will have to parse the labels string to extract the list of matching categories using `tf.strings.split`.

Object Detection

For object detection, the metadata file needs to include the bounding box of the object in the image. This can be accomplished by having a third column that contains the bounding box vertices in a predefined order (such as counterclockwise starting from top-left). For segmentation problems, this column will contain a polygon rather than a bounding box (see Figure 5-9).

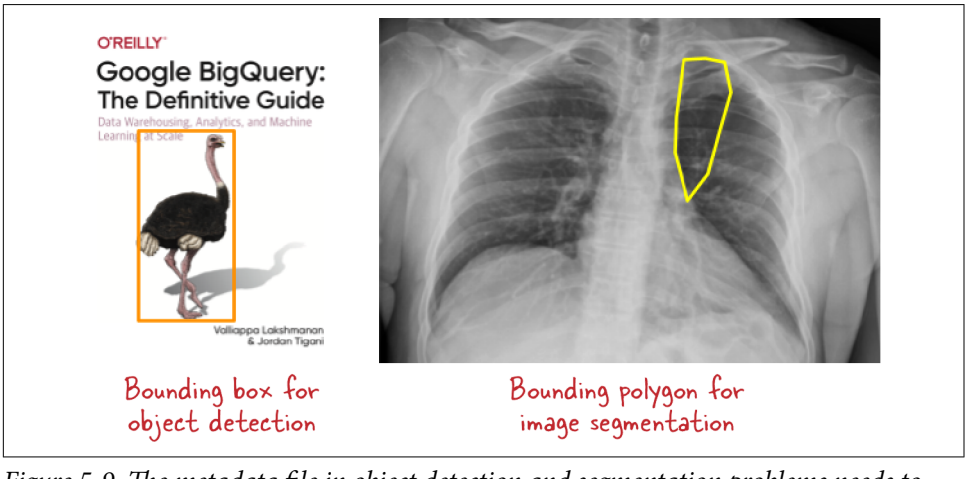


Figure 5-9. The metadata file in object detection and segmentation problems needs to include a bounding box or polygon, respectively.

Doughnut-shaped objects (with a center that is not part of the object) can be represented by a pair of polygons where the inner polygon has its vertices running in the opposite direction. To avoid this complexity, the segmentation boundaries are sometimes represented simply as a set of pixels instead of polygons.

Labeling at Scale

Manually labeling thousands of images is cumbersome and error-prone. How can we make it more efficient and accurate? One way is to use tools that make it possible to hand-label thousands of images efficiently. The other is to use methods to catch and correct labeling errors.

Labeling User Interface

A labeling tool should have a facility to display the image, and enable the rater to quickly select valid categories and save the rating to a database.

To support object identification and image segmentation use cases, the tool should have annotation capability and the ability to translate drawn bounding boxes or polygons into image pixel coordinates. The Computer Vision Annotation Tool (see Figure 5-10) is a free, web-based video and image annotation tool that is available online and can be installed locally. It supports a variety of annotation formats.

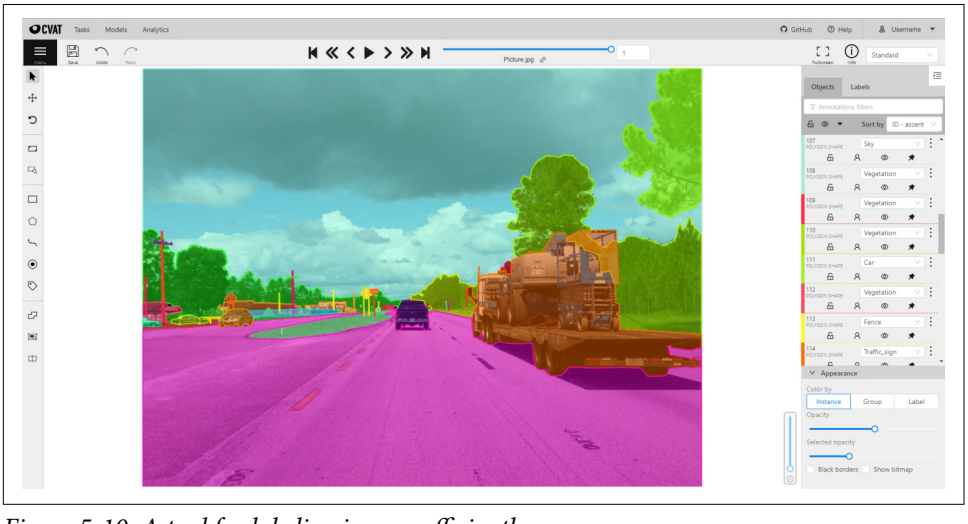


Figure 5-10. A tool for labeling images efficiently.

Multiple Tasks

Often, we will need to label images for multiple tasks. For example, we might need to classify the same images by flower type (daisy, tulip, ...), color (yellow, red, ...), location (indoors, outdoors, ...), planting style (potted, in-ground, …), and so on. An efficient approach in such situations is to do the labeling using the interactive functionality of a Jupyter notebook (see Figure 5-11).

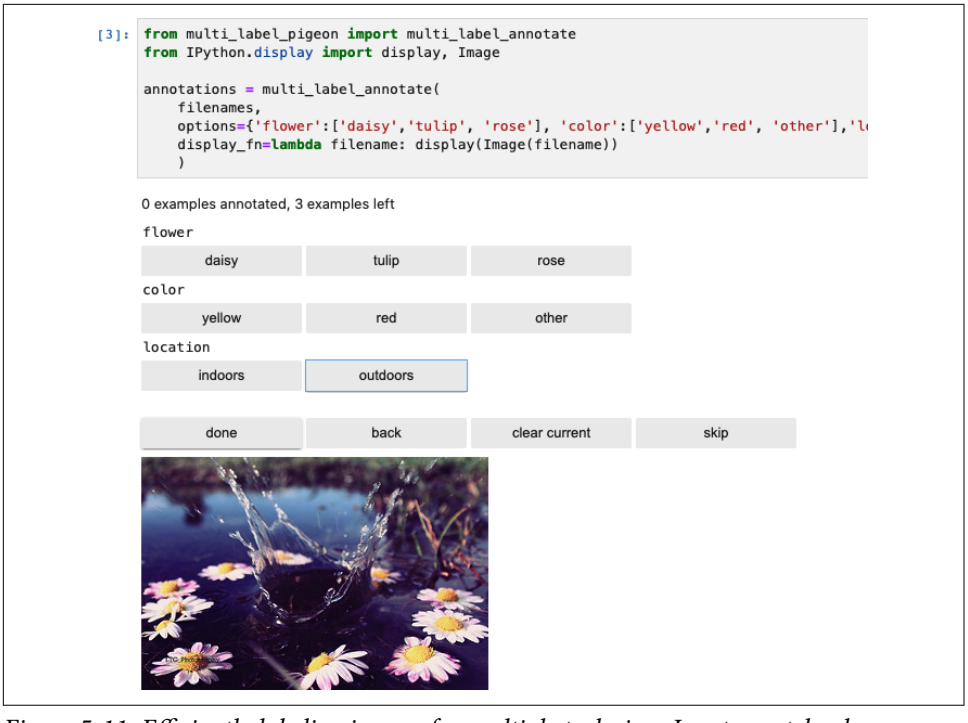


Figure 5-11. Efficiently labeling images for multiple tasks in a Jupyter notebook.

The functionality is provided by the Python package `multi-label-pigeon`:

```
annotations = multi_label_annotate(
    filenames,
    options={'flower':['daisy','tulip', 'rose'],
             'color':['yellow','red', 'other'],
             'location':['indoors','outdoors']},
    display_fn=lambda filename: display(Image(filename))
)
with open('label.json', 'w') as ofp:
    json.dump(annotations, ofp, indent=2)
```

The full code is in *05_label_images.ipynb* on GitHub for this book. The output is a JSON file with annotations for all the tasks for all the images:

```
{
    "flower_images/10172379554_b296050f82_n.jpg": {
        "flower": [
            "daisy"
        ],
        "color": [
            "red"
        ],
        "location": [
            "outdoors"
        ]
    },
```

Voting and Crowdsourcing

Manual labeling is subject to two challenges: human error and inherent uncertainty. Raters may get tired and wrongly identify an image. It may also be the case that the classification is ambiguous. Consider for example an X-ray image: radiologists may differ on whether something is a fracture or not.

In both these situations, it can be helpful to implement a voting system. For example, an image might be shown to two raters. If the raters agree, their label is assigned to the image. If the raters disagree, we can choose to do one of several things:

- Discard the image if we don't want to train using ambiguous data.
- Consider the image as belonging to a neutral class.
- Have the final label be determined by a third labeler, in effect making it a majority vote of three labelers. Of course, it is possible to increase the voting pool to any odd number.

Voting applies to multilabel problems as well. We just need to treat the incidence of each category as a binary classification problem, and then assign to an image all the labels on which a majority of raters agree.

Even object identification and segmentation boundaries can be determined via voting. An example of such a system is shown in Figure 5-12—the primary purpose of the CAPTCHA system is to identify whether a user is a robot or a human, but a secondary purpose is to crowdsource the labeling of images. It is clear that by decreasing the size of the tiles, it is possible to get a finer-grained labeling. By occasionally adding images or tiling, and collecting the results over many users, it is possible to get images successfully labeled.

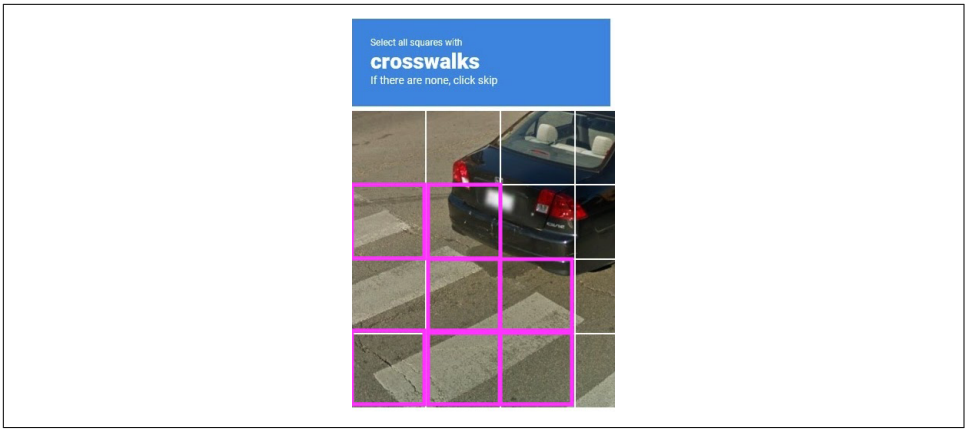


Figure 5-12. Crowdsourcing object detection or segmentation polygons.

Labeling Services

Even with efficient labeling, it can take days or months to label all the images needed to train a state-of-the-art image model. This is not a productive use of a data scientist's time. Because of this, many *labeling services* have cropped up. These are businesses that distribute the work of labeling images among dozens of employees at low-cost locations. Typically, we have to provide a few sample images and a description of the technique that needs to be employed to label images.

Labeling services are a bit more sophisticated than crowdsourcing. These services work not only for well-known objects (stop signs, crosswalks, etc.), but also for tasks where a layperson can be taught to make the correct decision quickly (e.g., a fracture versus a scratch mark in X-ray images). That said, you would probably not use labeling services for tasks like identifying the molecular structure of a virus that would take significant domain expertise.

Examples of labeling services include AI Platform Data Labeling Service, Clarifai, and Lionbridge. You'd typically work with the procurement department of your organization to use such a service. You should also verify how these services handle sensitive or personally identifying data.

Automated Labeling

In many situations, it is possible to obtain labels in an automated way. These methods can be useful even if they are not 100% accurate because it is far more efficient for raters to correct automatically obtained labels than it is for them to assign labels to images one by one.

Labels from Related Data

As an example, you might be able to obtain the label for an image by looking at the section of the product catalog that the image appears in, or by doing entity extraction from the words that describe the image.

In some cases, ground truth is available by looking at only a few of the pixels of the image. For example, seismic images can be labeled at locations where wells were dug and core samples extracted. A radar image may be labeled using the readings of ground rain gauges at locations where those gauges are installed. These point labels may be used to label tiles of the original image. Alternatively, the point labels may be spatially interpolated, and the spatially interpolated data used as labels for tasks such as segmentation.

Noisy Student

It is possible to stretch the labeling of images using a Noisy Student model. This approach works as follows:

- Manually label, say, 10,000 images.
- Use these images to train a small machine learning model. This is the *teacher model*.
- Use the trained machine learning model to predict the labels of, say, one million unlabeled images.
- Train a larger machine learning model, called the *student model*, on the combination of labeled and pseudo-labeled images. During the learning of the student model, employ dropout and random data augmentations (covered in Chapter 6) so that this model generalizes better than the teacher.
- Iterate by putting the student model back as the teacher.

It is possible to manually correct the pseudo-labels by choosing images where the machine learning models are not confident. This can be incorporated into the Noisy Student paradigm by doing the manual labeling before putting the student back as the new teacher model.

Self-Supervised Learning

In some cases, the machine learning approach can itself provide labels. For example, to create embeddings of images, we can train an autoencoder, as we will describe in Chapter 10. In an autoencoder, the image serves as its own label.

Another way that learning can be self-supervised is if the label will be known after some time. It may be possible to label a medical image based on the eventual outcome for the patient. If a patient subsequently develops emphysema, for example, that label

can be applied to lung images taken of the patient a few months prior to the diagnosis. Such a labeling method works for many forecasts of future activity: a satellite weather image might be labeled based on the subsequent occurrence of cloud-to-ground lightning as detected by a ground-based network, and a dataset for predicting whether a user is going to abandon their shopping cart or cancel their subscription can be labeled based on the eventual action of the user. So, even if our images don't have a label immediately at capture time, it can be worth holding on to them until we eventually get a label for them.

Many data quality problems can be framed in a self-supervised way. For example, if the task is to fill in an image of the ground when clouds are obstructing the view, the model can be trained by artificially removing parts of a clear-sky image and using the actual pixel values as labels.

Bias

The ideal machine learning dataset is one that allows us to train a model that will be perfect when it is placed into production. If, on the other hand, certain examples are under- or overrepresented in the dataset in such a way as to produce lower accuracy in those scenarios when they are encountered in production, then we have a problem. The dataset is then said to be *biased*.

In this section, we will discuss sources of dataset bias, how to collect data for a training dataset in an unbiased way, and how to detect bias in your dataset.

Sources of Bias

Bias in a dataset is a characteristic of the dataset that will lead to unwanted behavior when the model is placed into production.

We find that many people confuse two related, but separate, concepts: bias and imbalance. Bias is different from imbalance—it is quite possible that less than 1% of the pictures taken by an automatic wildlife camera are of jaguars. A dataset of wildlife pictures that has a very small proportion of jaguars is to be expected: it's unbalanced, but this is not evidence of bias. We might downsample commonly occurring animals and upsample unusual ones to help the machine learning model learn to identify different types of wildlife better, but such upsampling does not make the dataset biased. Instead, dataset bias is any aspect of the dataset that causes the model to behave in unwanted ways.

There are three sources of bias. *Selection bias* happens when the model is trained on a skewed subset of the scenarios that it will encounter in production. *Measurement bias* occurs when the way an image is collected varies between training and production. *Confirmation bias* happens when the distribution of values in real life leads to the

model reinforcing unwanted behaviors. Let's take a closer look at why each of these might occur; then we'll quickly show you how to detect bias in a dataset.

Selection Bias

Selection bias usually happens as a result of imperfect data collection—we mistakenly limit the source of our data, such that certain categories are excluded or poorly sampled. For example, suppose we're training a model to identify objects we sell. We might have trained the model on images in our product catalog, but this may have caused products from our partners to not be included. Therefore, the model will not recognize partner items that we sell but that were not in our product catalog. Similarly, an image model trained on photographs of houses found in county records may perform poorly on houses under construction if unfinished homes are not subject to county taxes, and are therefore not in the records.

A common reason for selection bias is that certain types of data are easier to collect than others. For example, it might be easier to collect images of French and Italian art than of Jamaican or Fijiian art. Datasets of artworks can therefore underrepresent certain countries or time periods. Likewise, previous years' product catalogs may be easy to find, but competitors' catalogs for this year might not be available yet, so our dataset might be up-to-date for our products but not for those of our competitors.

Sometimes selection bias happens simply because the training dataset is collected in a fixed time period, whereas the production time period is a lot more variable. For example, the training dataset might have been collected on a clear day, but the system is expected to work night and day, in clear weather and in rainy weather.

Selection bias can also happen as a result of outlier pruning and dataset cleanup. If we discard images of houseboats, barns, and mobile homes, the model will not be able to identify such buildings. If we are creating a dataset of seashells and discard any images of a shell with the animal still in it, then the model will perform poorly if shown a living crustacean.

To fix selection bias, it is necessary to work backward from the production system. What types of houses will need to be identified? Are there enough examples of such houses in the dataset? If not, the solution is to proactively collect such images.

Measurement Bias

Measurement bias occurs as a result of differences in the way an image is collected for training versus in production. These variations lead to systematic differences—perhaps we used a high-quality camera for the training images, but our production system employs an off-the-shelf camera that has a lower aperture, white balance, and/or resolution.

Measurement bias can also happen because of differences in who provides the data in training versus in production. For example, we may want to build a tool to help hikers identify wildflowers. If the training dataset consists of professional photographs, the photographs will include sophisticated effects like bokeh that will not be present in the photographs provided by a typical hiker for purposes of identification.

Measurement bias also happens when the labeling of images is done by a group of people. Different raters may have different standards, and inconsistencies in labeling can lead to poorer machine learning models.

Measurement bias can also be quite subtle. Perhaps all the photographs of foxes are taken against snow, whereas all the photographs of dogs are against grass. A machine learning model may learn to discriminate between snow and grass and achieve superior accuracy to a model that actually learns the features of foxes and dogs. So, we need to be mindful of what other things are within our images (and examine model explanations) to ensure our models learn the things we want them to learn.

Confirmation Bias

Remember when we said that many people confuse bias and imbalance? The difference and interrelationship between the two is particularly important when it comes to confirmation bias. A dataset may be biased even if it accurately represents an imbalanced real-world distribution—this is something that you should keep in mind as you read this section. Remember that bias in a dataset includes anything about the dataset that leads to unwanted behavior in ML models trained on that dataset.

Donald Rumsfeld, who was the US Secretary of Defense in 2002, famously listed three categories of knowledge:

> There are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns—the ones we don't know we don't know. And if one looks throughout the history of our country and other free countries, it is the latter category that tends to be the difficult one.

Confirmation bias is bias that we don't know about when we collect the data but which can nevertheless play havoc with models trained on the dataset. Human reluctance to examine the reasons why certain imbalances exist can lead to ML models perpetuating existing biases.

Collecting data "in the wild" can lead to confirmation bias. For example, at the time of writing, firefighters tend to be predominantly men. If we were to collect a random sample of images of firefighters, chances are that all the images would be of male firefighters. A machine learning model trained on such a dataset, when shown an image of a woman firefighter, might generate the caption that this is a woman in costume at a Halloween party. That would be pretty offensive, wouldn't it? This is a made-up

example, but it illustrates how an existing bias in society gets amplified when datasets reflect the real world. Do a search of recent news headlines about biased AI, and you will find any number of real-world disasters that have, at their core, a similar bias because they reflect real-world distributions.

A small-town newspaper will tend to cover events that occur in the town, and by virtue of this data being "in the wild," most of the photographs of concerts, fairs, and outdoor dining will contain images of the majority community. On the other hand, most of the photographs of minority-community teens that appear in the newspaper might be photographs of arrests. Arrest photographs of majority-community teens will also appear in the newspaper, but they will be greatly outnumbered by photographs of those teens in outdoor settings. Given such a dataset, a machine learning model will learn to associate minority-community members with jail and majority-community members with benign activities. Again, this is an example of a model confirming and perpetuating the bias of the newspaper editors because of what newspaper stories tend to cover.

Confirmation bias can also amplify existing biases in terms of labels. If a company trains a model to sort through job applications it has received and classify them based on who finally got hired, the model will learn any bias (whether it is in favor of elite colleges or against minority candidates) that the company's current interviewers have. If the company tends to hire very few Black candidates or highly favors Ivy League candidates, the model will learn and replicate that. The "unbiased" model has become extremely biased.

To address confirmation bias, we have to be aware of this blind spot that we have, and consciously move areas of unknown unknowns to one of the other two categories. We have to be aware of the existing biases in our company, in our industry, or in society and carefully validate that our dataset is not collected in such a way as to amplify that bias. The recommended approach involves both awareness (of potential bias) and active data collection (to mitigate this bias).

Detecting Bias

To detect bias, you can carry out *sliced evaluations*—essentially, compute the objective function of your model, but only on members of a group. Compare this with the value of the metric for non-members of the group. Then investigate any groups for which the sliced metrics are very different from that of the overall dataset. You can also apply a Bayesian approach and calculate measures such as "what are the chances that a retinal scan will be categorized as diseased if the sample is from a racial minority?"

The Aequitas Fairness Tree approach suggests which metric to monitor depending on whether the ML model is used punitively or assistively.

Creating a Dataset

Once we have collected a set of images and labeled them, we are ready to train an ML model with those images. However, we will have to split the dataset into three parts: training, validation, and testing sets. We will also want to take the opportunity to store the image data in a more efficient format for ML. Let's look at both of these steps and how our training program would read files in this format.

Splitting Data

The dataset of images and labels will have to be split into three parts, for training, validation, and testing. The actual proportion is up to us, but something like 80:10:10 is common.

The training dataset is the set of examples that are presented to the model. The optimizer uses these examples to tune the weights of the model so as to reduce the error, or *loss*, on the training dataset. The loss on the training dataset at the end of training is not, however, a reliable measure of the performance of the model. To estimate that, we have to use a dataset of examples that have not been shown to the model during its training process. That is the purpose of the validation dataset.

If we were training only one model, and training it only once, we would need only the training and validation datasets (in this case, an 80:20 split is common). However, it is very likely that we will retry the training with a different set of hyperparameters —perhaps we will change the learning rate, or decrease the dropout, or add a couple more layers to the model. The more of these hyperparameters we optimize against the validation dataset, the more the skill of the model on the validation dataset gets incorporated into the structure of the model itself. The validation dataset is thus no longer a reliable estimate of how the model will perform when given net new data.

Our final evaluation (of the model fit on the training dataset, and using parameters optimized on the validation dataset) is carried out on the test dataset.

Splitting the dataset at the beginning of each training run is not a good idea. If we do this, every experiment will have different training and validation datasets, which defeats the purpose of retaining a truly independent test dataset. Instead, we should split once, and then continue using the same training and validation datasets for all our hyperparameter tuning experiments. Therefore, we should save the training, validation, and test CSV files and use these consistently throughout the model lifecycle.

Sometimes, we might want to do cross-validation on the dataset. To do this, we train the model multiple times using different splits of the first 90% into training and validation datasets (the test dataset remains the same 10%). In such a case, we'd write out multiple training and validation files. Cross-validation is common on small datasets,

but much less common in machine learning on large datasets such as those used in image models.

TensorFlow Records

The CSV file format mentioned in the previous section is not recommended for large-scale machine learning because it relies on storing image data as individual JPEG files, which is not very efficient. A more efficient data format to use is TensorFlow Records (TFRecords). We can convert our JPEG image files into TFRecords using Apache Beam.

First, we define a method to create a TFRecord given the image filename and the label of the image:

```
def create_tfrecord(filename, label, label_int):
    img = read_and_decode(filename)
    dims = img.shape
    img = tf.reshape(img, [-1])  # flatten to 1D array
    return tf.train.Example(features=tf.train.Features(feature={
        'image': _float_feature(img),
        'shape': _int64_feature([dims[0], dims[1], dims[2]]),
        'label': _string_feature([label]),
        'label_int': _int64_feature([label_int])
    })).SerializeToString()
```

The TFRecord is a dictionary with two main keys: `image` and `label`. Because different images can have different sizes, we also take care to store the shape of the original image. To save time looking up the index of the label during training, we also store the label as an integer.

Besides efficiency, TFRecords give us the ability to embed image metadata such as the label, bounding box, and even additional ML inputs such as the location and timestamp of the image as part of the data itself. This way, we don't need to rely on ad hoc mechanisms such as the file/directory name or external files to encode metadata.

The image itself is a flattened array of floating-point numbers—for efficiency, we are doing the JPEG decoding and scaling before writing to the TFRecords. This way, it is not necessary to redo these operations as we iterate over the training dataset:

```
def read_and_decode(filename):
    img = tf.io.read_file(filename)
    img = tf.image.decode_jpeg(img, channels=IMG_CHANNELS)
    img = tf.image.convert_image_dtype(img, tf.float32)
    return img
```

As well as being more efficient, decoding the images and scaling their values to [0, 1] before writing them out to TFRecords has two other advantages. First, this puts the data in the exact form required by the image models in TensorFlow Hub (see Chapter 3). Second, it allows the reading code to use the data without having to know whether the files were in JPEG or PNG or some other image format.

An equally valid approach is to store the data in the TFRecord as JPEG bytes, rely on TensorFlow's `decode_image()` function to read the data, and scale the image values to [0, 1] in the preprocessing layer of the model. Because the JPEG bytes are compressed using an algorithm tailored for images, the resulting files can be smaller than gzipped TFRecord files consisting of raw pixel values. Use this approach if bandwidth is more important than decoding time. Another benefit of this approach is that the decoding operation is usually pipelined on the CPU while the model trains on a GPU, so it might be essentially free.

The Apache Beam pipeline consists of getting the training, validation, and test CSV files, creating TFRecords, and writing the three datasets with appropriate prefixes. For example, the training TFRecord files are created using:

```
with beam.Pipeline() as p:
    (p
     | 'input_df' >> beam.Create(train.values)
     | 'create_tfr' >> beam.Map(lambda x: create_tfrecord(
            x[0], x[1], LABELS.index(x[1])))
     | 'write' >> beam.io.tfrecordio.WriteToTFRecord(
            'output/train', file_name_suffix='.gz')
    )
```

While there are several advantages to decoding and scaling the pixel values before writing them out to TFRecords, the floating-point pixel data tends to take up more space than the original byte stream. This drawback is addressed in the preceding code by compressing the TFRecord files. The TFRecord writer will automatically compress the output files when we specify that the filename suffix should be *.gz*.

Running at scale

The previous code is fine for transforming a few images, but when you have thousands to millions of images, you'll want a more scalable, resilient solution. The solution needs to be fault-tolerant, able to be distributed to multiple machines, and capable of being monitored using standard DevOps tools. Typically, we'd also want to pipe the output to a cost-efficient blob storage as new images come streaming in. Ideally, we'd want this to be done in a serverless way so that we don't have to manage and scale up/down this infrastructure ourselves.

One solution that addresses these production needs of resilience, monitoring, streaming, and autoscaling is to run our Apache Beam code on Google Cloud Dataflow rather than in a Jupyter notebook:

```
with beam.Pipeline('DataflowRunner', options=opts) as p:
```

The options can be obtained from the command line using standard Python constructs (like `argparse`) and will typically include the Cloud project to be billed and the Cloud region in which to run the pipeline. Besides Cloud Dataflow, other runners for Apache Beam include Apache Spark and Apache Flink.

As long as we are creating a pipeline like this, it can be helpful to capture all the steps of our workflow within it, including the step of splitting the dataset. We can do this as follows (the full code is in *jpeg_to_tfrecord.py* on GitHub):

```
with beam.Pipeline(RUNNER, options=opts) as p:
    splits = (p
              | 'read_csv' >> beam.io.ReadFromText(arguments['all_data'])
              | 'parse_csv' >> beam.Map(lambda line: line.split(','))
              | 'create_tfr' >> beam.Map(lambda x: create_tfrecord(
                    x[0], x[1], LABELS.index(x[1])))
              | 'assign_ds' >> beam.Map(assign_record_to_split)
             )
```

where the `assign_record_to_split()` function assigns each record to one of the three splits:

```
def assign_record_to_split(rec):
    rnd = np.random.rand()
    if rnd < 0.8:
        return ('train', rec)
    if rnd < 0.9:
        return ('valid', rec)
    return ('test', rec)
```

At this point, splits consists of tuples like:

```
('valid', 'serialized-tfrecord...')
```

These can then be farmed out into three sets of sharded files with the appropriate prefixes:

```
for s in ['train', 'valid', 'test']:
    _ = (splits
         | 'only_{}'.format(s) >> beam.Filter(lambda x: x[0] == s)
         | '{}_records'.format(s) >> beam.Map(lambda x: x[1])
         | 'write_{}'.format(s) >> beam.io.tfrecordio.WriteToTFRecord(
               os.path.join(OUTPUT_DIR, s), file_name_suffix='.gz')
        )
```

When this program is run, the job will be submitted to the Cloud Dataflow service, which will execute the entire pipeline (see Figure 5-13) and create TFRecord files corresponding to all three splits with names like *valid-00000-of-00005.gz*.

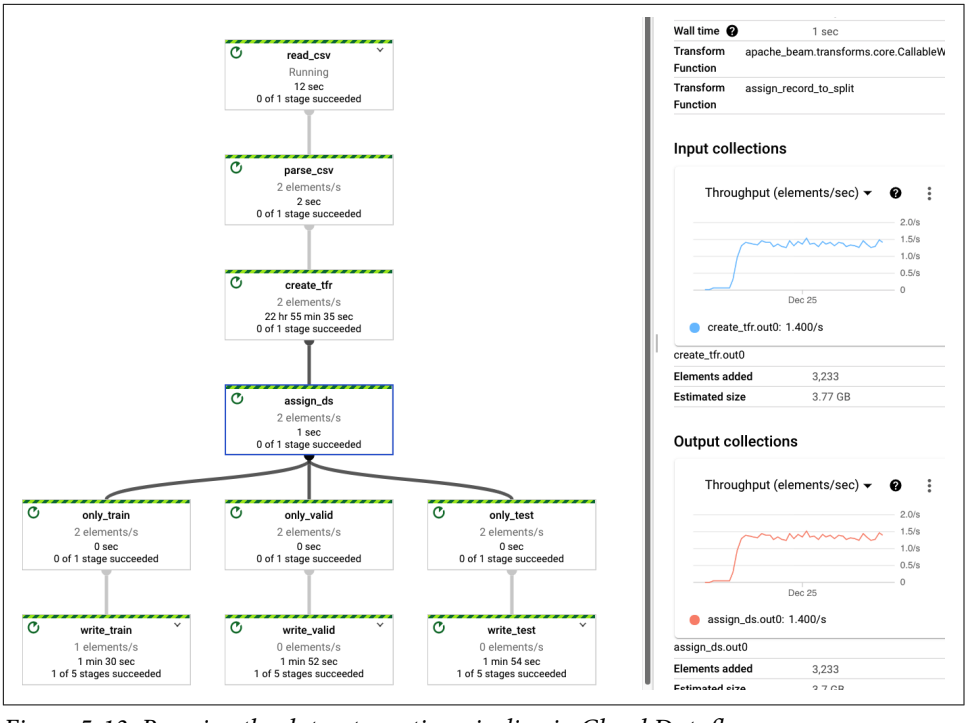


Figure 5-13. Running the dataset creation pipeline in Cloud Dataflow.

Balancing Flexibility and Maintainability

Because ML training is so computationally expensive, it can be worth the duplication in storage to create efficient, ready-to-train-on datasets for your ML projects. There are two drawbacks, however. First, each of your ML projects will typically involve different preprocessing steps (more on this in Chapter 6), and creating separate datasets for each project can lead to increased storage costs. That's why, in this chapter, we did something intermediate—while we decoded the JPEG images and scaled them to be in the range [0, 1], we left them in the original size. Resizing is one of the preprocessing steps that will be done in the training data pipeline because each ML project will tend to want to resize the images differently.

The second drawback is that such extracted datasets run the risk of running afoul of the data governance policies in place at your organization. There might be regulatory and compliance risks associated with extracting image data from the original data lake (where presumably all access is logged and monitored, and data is appropriately aged off) and storing intermediate files or sample metadata, which are potentially harder to track and might need to be saved in different and less-governed storage locations (like hard disks of a cluster for fast I/O during the ML process).

Changing the input from CSV files to Cloud pub/sub will convert this pipeline from a batch pipeline to a streaming one. All the intermediate steps remain the same, and the resulting sharded TFRecords (which are in a format conducive for machine learning) can function as our ML *data lake*.

TensorFlow Recorder

In the previous sections we looked at how to manually create TFRecord files, carrying out some extract, transform, load (ETL) operations along the way. If you already have data in Pandas or CSV files, it may be much more convenient to use the TFRecorder Python package, which adds a `tensorflow.to_tfr()` method to a Pandas dataframe:

```
import pandas as pd
import tfrecorder
csv_file = './all_data_split.csv'
df = pd.read_csv(csv_file, names=['split', 'image_uri', 'label'])
df.tensorflow.to_tfr(output_dir='gs://BUCKET/data/output/path')
```

The CSV files in this example are assumed to have lines that look like this:

```
valid,gs://BUCKET/img/abc123.jpg,daisy
train,gs://BUCKET/img/def123.jpg,tulip
```

TFRecorder will serialize the images into TensorFlow Records.

Running TFRecorder at scale in Cloud Dataflow involves adding a few parameters to the call:

```
df.tensorflow.to_tfr(
    output_dir='gs://my/bucket',
    runner='DataflowRunner',
    project='my-project',
    region='us-central1',
    tfrecorder_wheel='/path/to/my/tfrecorder.whl')
```

For details on how to create and load a wheel to be used, please check the TFRecorder documentation.

Reading TensorFlow Records

To read TensorFlow Records, use a `tf.data.TFRecordDataset`. To read all the training files into a TensorFlow dataset, we can pattern match and then pass the resulting files into `TFRecordDataset()`:

```
train_dataset = tf.data.TFRecordDataset(
    tf.data.Dataset.list_files(
        'gs://practical-ml-vision-book/flowers_tfr/train-*')
    )
```

The full code is in *06a_resizing.ipynb* on GitHub, but in a notebook in the folder for Chapter 6 because that's when we actually have to read these files.

The dataset at this point contains protobufs. We need to parse the protobufs based on the schema of the records that we wrote to the files. We specify that schema as follows:

```
feature_description = {
    'image': tf.io.VarLenFeature(tf.float32),
    'shape': tf.io.VarLenFeature(tf.int64),
    'label': tf.io.FixedLenFeature([], tf.string,
default_value=''),
        'label_int': tf.io.FixedLenFeature([], tf.int64, default_value=0),
}
```

Compare this with the code we used to create the TensorFlow Record:

```
return tf.train.Example(features=tf.train.Features(feature={
    'image': _float_feature(img),
    'shape': _int64_feature([dims[0], dims[1], dims[2]]),
    'label': _string_feature(label),
    'label_int': _int64_feature([label_int])
}))
```

The `label` and `label_int` have a fixed length (1), but the `image` and its `shape` are variable length (since they are arrays).

Given the proto and the feature description (or schema), we can read in the data using the function `parse_single_example()`:

```
rec = tf.io.parse_single_example(proto, feature_description)
```

For storage efficiency, variable-length arrays are stored as sparse tensors (see "What Is a Sparse Tensor?" on page 206). We can make them dense and reshape the flattened image array into a 3D tensor, giving us the full parsing function:

```
def parse_tfr(proto):
    feature_description = ...
    rec = tf.io.parse_single_example(proto, feature_description)
    shape = tf.sparse.to_dense(rec['shape'])
    img = tf.reshape(tf.sparse.to_dense(rec['image']), shape)
    return img, rec['label_int']
```

We can now apply the parsing function to every proto that is read using `map()`:

```
train_dataset = tf.data.TFRecordDataset(
    [filename for filename in tf.io.gfile.glob(
        'gs://practical-ml-vision-book/flowers_tfr/train-*')
    ]).map(parse_tfr)
```

At this point, the training dataset gives us the image and its label, which we can use just like the image and label we obtained from the CSV dataset in Chapter 2.

What Is a Sparse Tensor?

A *sparse tensor* is an efficient representation of tensors that have only a few nonzero values. If a tensor has many zeros, it is more efficient to store the tensor if we represent only the nonzero values. Consider a 2D tensor that has many zeros:

```
[[0, 0, 3, 0, 0, 0, 0, 5, 0, 0],
 [0, 2, 0, 0, 0, 0, 4, 0, 0, 0]]
```

Instead of storing this tensor with 10 numbers, we could represent it as follows:

- The nonzero values in the tensor: `[3, 5, 2, 4]`
- The indices of those nonzero values: `[[0, 2], [0, 7], [1, 1], [1, 6]]`
- The dense shape of the tensor: `(2, 10)`

TensorFlow does exactly this: it represents a sparse tensor as three separate dense tensors, `indices`, `values`, and `dense_shape`. It also supports direct computations on the sparse tensors, which are more efficient than the equivalent computations on dense tensors. For example, if we call `tf.sparse.maximum()` on our example sparse tensor it will need to iterate through only the values tensor (4 numbers), whereas if we called `tf.math.maximum()` it would have to iterate through the full 2D tensor (20 values).

Summary

In this chapter, we looked at how to create vision datasets consisting of images and the labels associated with those images. The images can be photographs or can be produced by sensors that create 2D or 3D projections. It is possible to align several such images into a single image by treating the individual image's values as channels.

Image labeling often has to be done manually, at least in the beginning stages of a project. We looked at different types of labels for different problem types, how to organize the labels, how to efficiently label images, and how to use voting to reduce errors in the labels. Labels can sometimes be extracted automatically from the eventual outcome, or from ancillary datasets. It is also possible to set up an iterative Noisy Student process to create pseudo-labels.

We also discussed dataset bias, the causes of bias, and how to lower the chances of bias in our datasets. We will look at how to diagnose bias in Chapter 8.

Finally, we saw how to create training and validation, test splits of our data, and store these three image datasets efficiently in a data lake. In the next two chapters, you will learn how to train ML models on the datasets you created for that purpose. In Chapter 6, we will explore how to preprocess images for machine learning, and in Chapter 7 we will discuss how to train ML models on the preprocessed images.

CHAPTER 6

Preprocessing

In Chapter 5, we looked at how to create training datasets for machine learning. This is the first step of the standard image processing pipeline (see Figure 6-1). The next stage is preprocessing the raw images in order to feed them into the model for training or inference. In this chapter, we will look at why images need to be preprocessed, how to set up preprocessing to ensure reproducibility in production, and ways to implement a variety of preprocessing operations in Keras/TensorFlow.

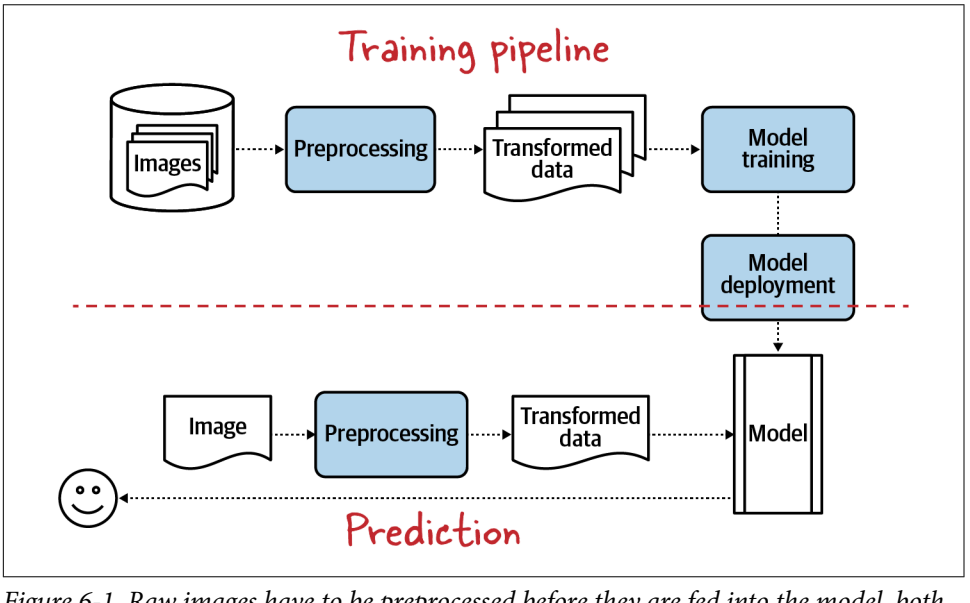


Figure 6-1. Raw images have to be preprocessed before they are fed into the model, both during training (top) and during prediction (bottom).

The code for this chapter is in the *06_preprocessing* folder of the book's GitHub repository. We will provide file names for code samples and notebooks where applicable.

Reasons for Preprocessing

Before raw images can be fed into an image model, they usually have to be preprocessed. Such preprocessing has several overlapping goals: shape transformation, data quality, and model quality.

Shape Transformation

The input images typically have to be transformed into a consistent size. For example, consider a simple DNN model:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(512, 256, 3)),
    tf.keras.layers.Dense(128,
                          activation=tf.keras.activations.relu),
    tf.keras.layers.Dense(len(CLASS_NAMES), activation='softmax')
])
```

This model requires that the images fed into it are 4D tensors with an inferred batch size, 512 columns, 256 rows, and 3 channels. Every layer that we have considered so far in this book needs a shape to be specified at construction. Sometimes the specification can be inferred from previous layers, and does not have to be explicit: the first `Dense` layer takes the output of the `Flatten` layer and therefore is built to have 512 * 256 * 3 = 393,216 input nodes in the network architecture. If the raw image data is not of this size, then there is no way to map each input value to the nodes of the network. So, images that are not of the right size have to be transformed into tensors with this exact shape. Any such transformation will be carried out in the preprocessing stage.

Data Quality Transformation

Another reason to do preprocessing is to enforce data quality. For example, many satellite images have a terminator line (see Figure 6-2) because of solar lighting or the Earth's curvature.

Solar lighting can lead to different lighting levels in different parts of the image. Since the terminator line moves throughout the day and its location is known precisely from the timestamp, it can be helpful to normalize each pixel value taking into account the solar illumination that the corresponding point on the Earth receives. Or, due to the Earth's curvature and the point of view of the satellite, there might be parts of the images that were not sensed by the satellite. Such pixels might be masked or

assigned a value of `-inf`. In the preprocessing step, it is necessary to handle this somehow because neural networks will expect to see a finite floating-point value; one option is to replace these pixels with the mean value in the image.

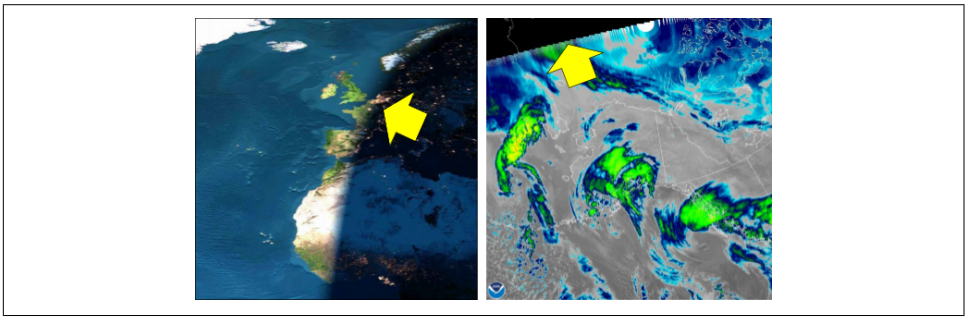


Figure 6-2. Impact of solar lighting (left) and Earth's curvature (right). Images from NASA © Living Earth and the NOAA GOES-16 satellite.

Even if your dataset doesn't consist of satellite imagery, it's important to be aware that data quality problems, like the ones described here for satellite data, pop up in many situations. For example, if some of your images are darker than others, you might want to transform the pixel values within the images to have a consistent white balance.

Improving Model Quality

A third goal of preprocessing is to carry out transformations that help improve the accuracy of models trained on the data. For example, machine learning optimizers work best when data values are small numbers. So, in the preprocessing stage, it can be helpful to scale the pixel values to lie in the range [0, 1] or [–1, 1].

Some transformations can help improve model quality by increasing the effective size of the dataset that the model was trained on. For example, if you are training a model to identify different types of animals, an easy way to double the size of your dataset is to *augment* it by adding flipped versions of the images. In addition, adding random perturbations to images results in more robust training as it limits the extent to which the model overfits.

Of course, we have to be careful when applying left-to-right transformations. If we are training a model with images that contain a lot of text (such as road signs), augmenting images by flipping them left to right would reduce the ability of the model to recognize the text. Also, sometimes flipping the images can destroy information that we require. For example, if we are trying to identify products in a clothing store, flipping images of buttoned shirts left to right may destroy information. Men's shirts have the button on the wearer's right and the button hole on the wearer's left, whereas women's shirts are the opposite. Flipping the images randomly would make it

impossible for the model to use the position of the buttons to determine the gender the clothes were designed for.

Size and Resolution

As discussed in the previous section, one of the key reasons to preprocess images is to ensure that the image tensors have the shape expected by the input layer of the ML model. In order to do this, we usually have to change the size and/or resolution of the images being read in.

Consider the flower images that we wrote out into TensorFlow Records in Chapter 5. As explained in that chapter, we can read those images using:

```
train_dataset = tf.data.TFRecordDataset(
    [filename for filename in tf.io.gfile.glob(
        'gs://practical-ml-vision-book/flowers_tfr/train-*')
    ]).map(parse_tfr)
```

Let's display five of those images:

```
for idx, (img, label_int) in enumerate(train_dataset.take(5)):
    print(img.shape)
    ax[idx].imshow((img.numpy()));
```

As is clear from Figure 6-3, the images all have different sizes. The second image, for example (240x160), is in portrait mode, whereas the third image (281x500) is horizontally elongated.

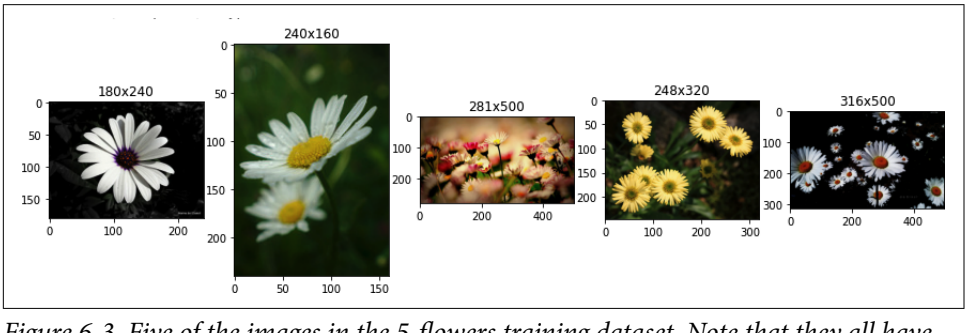


Figure 6-3. Five of the images in the 5-flowers training dataset. Note that they all have different dimensions (marked on top of the image).

Using Keras Preprocessing Layers

When the input images are of different sizes, we need to preprocess them to the shape expected by the input layer of the ML model. We did this in Chapter 2 using a TensorFlow function when we read the images, specifying the desired height and width:

```
img = tf.image.resize(img, [IMG_HEIGHT, IMG_WIDTH])
```

Keras has a preprocessing layer called `Resizing` that offers the same functionality. Typically we will have multiple preprocessing operations, so we can create a Sequential model that contains all of those operations:

```
preproc_layers = tf.keras.Sequential([
    tf.keras.layers.experimental.preprocessing.Resizing(
        height=IMG_HEIGHT, width=IMG_WIDTH,
        input_shape=(None, None, 3))
    ])
```

To apply the preprocessing layer to our images, we could do:

```
train_dataset.map(lambda img: preproc_layers(img))
```

However, this won't work because the `train_dataset` provides a tuple (img, label) where the image is a 3D tensor (height, width, channels) while the Keras Sequential model expects a 4D tensor (batchsize, height, width, channels).

The simplest solution is to write a function that adds an extra dimension to the image at the first axis using `expand_dims()` and removes the batch dimension from the result using `squeeze()`:

```
def apply_preproc(img, label):
    # add to a batch, call preproc, remove from batch
    x = tf.expand_dims(img, 0)
    x = preproc_layers(x)
    x = tf.squeeze(x, 0)
    return x, label
```

With this function defined, we can apply the preprocessing layer to our tuple using:

```
train_dataset.map(apply_preproc)
```

Normally, we don't have to call `expand_dims()` and `squeeze()` in a preprocessing function because we apply the preprocessing function after a `batch()` call. For example, we would normally do:

```
train_dataset.batch(32).map(apply_preproc)
```

Here, however, we can't do this because the images that come out of the `train_dataset` are all of different sizes. To solve this problem, we can add an extra dimension as shown or use ragged batches.

The result is shown in Figure 6-4. Notice that all the images are now the same size, and because we passed in 224 for the `IMG_HEIGHT` and `IMG_WIDTH`, the images are squares. Comparing this with Figure 6-3, we notice that the second image has been squashed vertically whereas the third image has been squashed in the horizontal dimension and stretched vertically.

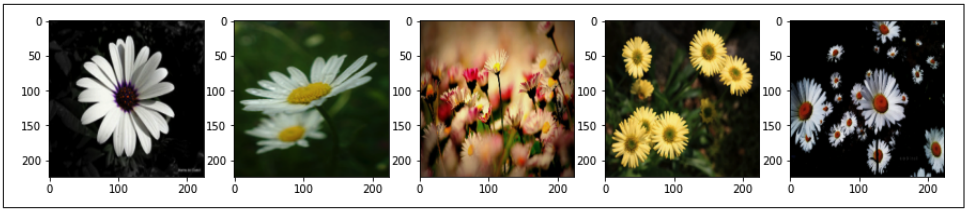


Figure 6-4. The effect of resizing the images to a shape of (224, 224, 3). Intuitively, stretching and squashing flowers will make them harder to recognize, so we would like to preserve the aspect ratio of the input images (the ratio of height to width). Later in this chapter, we will look at other preprocessing options that can do this.

The Keras `Resizing` layer offers several interpolation options when doing the squashing and stretching: `bilinear`, `nearest`, `bicubic`, `lanczos3`, `gaussian`, and so on. The default interpolation scheme (`bilinear`) retains local structures, whereas the `gaussian` interpolation scheme is more tolerant of noise. In practice, however, the differences between different interpolation methods are pretty minor.

The Keras preprocessing layers have an advantage that we will delve deeper into later in this chapter—because they are part of the model, they are automatically applied during prediction. Choosing between doing preprocessing in Keras or in TensorFlow thus often comes down to a trade-off between efficiency and flexibility; we will expand upon this later in the chapter.

Using the TensorFlow Image Module

In addition to the `resize()` function that we used in Chapter 2, TensorFlow offers a plethora of image processing functions in the `tf.image` module. We used `decode_jpeg()` from this module in Chapter 5, but TensorFlow also has the ability to decode PNG, GIF, and BMP and to convert images between color and grayscale. There are methods to work with bounding boxes and to adjust contrast, brightness, and so on.

In the realm of resizing, TensorFlow allows us to retain the aspect ratio when resizing by cropping the image to the desired aspect ratio and stretching it:

```
img = tf.image.resize(img, [IMG_HEIGHT, IMG_WIDTH],
                      preserve_aspect_ratio=True)
```

or padding the edges with zeros:

```
img = tf.image.resize_with_pad(img, [IMG_HEIGHT, IMG_WIDTH])
```

We can apply this function directly to each (img, label) tuple in the dataset as follows:

```
def apply_preproc(img, label):
    return (tf.image.resize_with_pad(img, 2*IMG_HEIGHT, 2*IMG_WIDTH),
            label)
train_dataset.map(apply_preproc)
```

The result is shown in Figure 6-5. Note the effect of padding in the second and third panels in order to avoid stretching or squashing the input images while providing the desired output size.

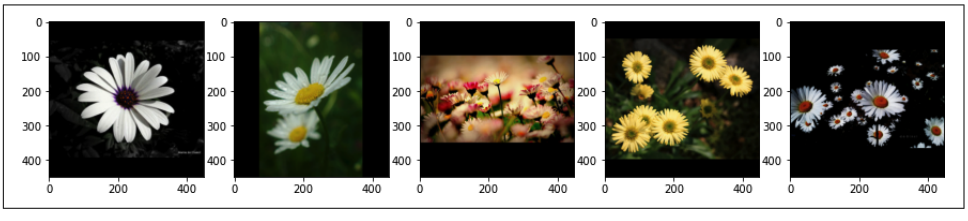


Figure 6-5. Resizing the images to (448, 448) with padding.

The eagle-eyed among you may have noticed that we resized the images to be larger than the desired height and width (twice as large, actually). The reason for this is that it sets us up for the next step.

While we've preserved the aspect ratio by specifying a padding, we now have padded images with black borders. This is not desirable either. What if we now do a "center crop"—i.e., crop these images (which are larger than what we want anyway) in the center?

Mixing Keras and TensorFlow

A center-cropping function is available in TensorFlow, but to keep things interesting, let's mix TensorFlow's `resize_with_pad()` and Keras's `CenterCrop` functionality.

In order to call an arbitrary set of TensorFlow functions as part of a Keras model, we wrap the function(s) inside a Keras `Lambda` layer:

```
tf.keras.layers.Lambda(lambda img:
                       tf.image.resize_with_pad(
                           img, 2*IMG_HEIGHT, 2*IMG_WIDTH))
```

Here, because we want to do the resize and follow it by a center crop, our preprocessing layers become:

```
preproc_layers = tf.keras.Sequential([
    tf.keras.layers.Lambda(lambda img:
                           tf.image.resize_with_pad(
                               img, 2*IMG_HEIGHT, 2*IMG_WIDTH),
                           input_shape=(None, None, 3)),
    tf.keras.layers.experimental.preprocessing.CenterCrop(
        height=IMG_HEIGHT, width=IMG_WIDTH)
    ])
```

Note that the first layer (`Lambda`) carries an `input_shape` parameter. Because the input images will be of different sizes, we specify the height and width as `None`, which leaves the values to be determined at runtime. However, we do specify that there will always be three channels.

The result of applying this preprocessing is shown in Figure 6-6. Note how the aspect ratio of the flowers is preserved and all the images are 224x224.

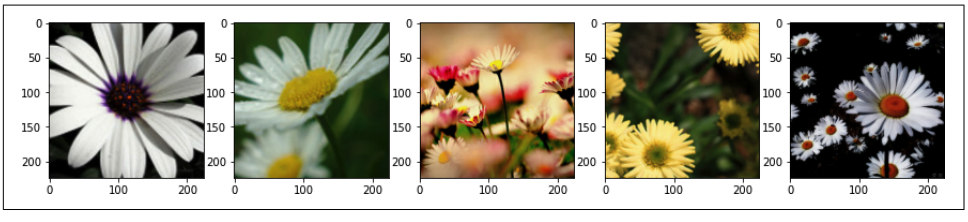


Figure 6-6. The effect of applying two processing operations: a resize with pad followed by a center crop.

At this point, you have seen three different places to carry out preprocessing: in Keras, as a preprocessing layer; in TensorFlow, as part of the `tf.data` pipeline; and in Keras, as part of the model itself. As mentioned earlier, choosing between these comes down to a trade-off between efficiency and flexibility; we'll explore this in more detail later in this chapter.

Model Training

Had the input images all been the same size, we could have incorporated the preprocessing layers into the model itself. However, because the input images vary in size, they cannot be easily batched. Therefore, we will apply the preprocessing in the ingest pipeline before doing the batching:

```
train_dataset = tf.data.TFRecordDataset(
    [filename for filename in tf.io.gfile.glob(
        'gs://practical-ml-vision-book/flowers_tfr/train-*')
    ]).map(parse_tfr).map(apply_preproc).batch(batch_size)
```

The model itself is the same MobileNet transfer learning model that we used in Chapter 3 (the full code is in *06a_resizing.ipynb* on GitHub):

```
layers = [
    hub.KerasLayer(
        "https://tfhub.dev/.../mobilenet_v2/...",
        input_shape=(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS),
        trainable=False,
        name='mobilenet_embedding'),
    tf.keras.layers.Dense(num_hidden,
                          activation=tf.keras.activations.relu,
                          name='dense_hidden'),
```

```
    tf.keras.layers.Dense(len(CLASS_NAMES),
                          activation='softmax',
                          name='flower_prob')
]
model = tf.keras.Sequential(layers, name='flower_classification')
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=lrate),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(
                  from_logits=False),
              metrics=['accuracy'])
history = model.fit(train_dataset, validation_data=eval_dataset, epochs=10)
```

The model training converges and the validation accuracy plateaus at 0.85 (see Figure 6-7).

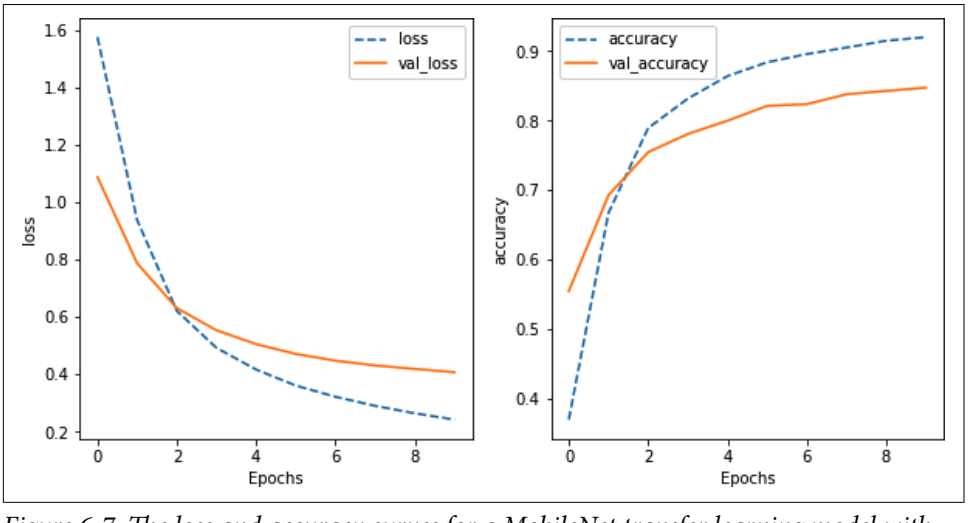


Figure 6-7. The loss and accuracy curves for a MobileNet transfer learning model with preprocessed layers as input.

Comparing Figure 6-7 against Figure 3-3, it seems that we have fared worse with padding and center cropping than with the naive resizing we did in Chapter 3. Even though the validation datasets are different in the two cases, and so the accuracy numbers are not directly comparable, the difference in accuracy (0.85 versus 0.9) is large enough that it is quite likely that the Chapter 6 model is worse than the Chapter 3 one. Machine learning is an experimental discipline, and we would not have known this unless we tried. It's quite possible that on a different dataset, fancier preprocessing operations will improve the end result; you have to try multiple options to figure out which method works best for your dataset.

Some prediction results are shown in Figure 6-8. Note that the input images all have a natural aspect ratio and are center cropped.

Figure 6-8. Images as input to the model, and predictions on those images.

Training-Serving Skew

During inference, we need to carry out the exact same set of operations on the image that we did during training (see Figure 6-1). Recall that we did preprocessing in three places:

1. When creating the file. When we wrote out the TensorFlow Records in Chapter 5, we decoded the JPEG files and scaled the input values to [0, 1].
2. When reading the file. We applied the function `parse_tfr()` to the training dataset. The only preprocessing this function did was to reshape the image tensor to [height, width, 3], where the height and width are the original size of the image.
3. In the Keras model. We then applied `preproc_layers()` to the images. In the last version of this method, we resized the images with padding to 448x448 and then center cropped them to 224x224.

In the inference pipeline, we have to perform all those operations (decoding, scaling, reshaping, resizing, center cropping) on the images provided by clients.[1] If we were to miss an operation or carry it out slightly differently between training and inference, it would cause potentially incorrect results. The condition where the training and inference pipelines diverge (therefore creating unexpected or incorrect behavior during inference not seen during training) is called *training-serving skew*. In order to prevent training-serving skew, it is ideal if we can reuse the exact same code both in training and for inference.

Broadly, there are three ways that we can set things up so that all the image preprocessing done during training is also done during inference:

1 Reality is more complex. There might be data preprocessing (e.g., data augmentation, covered in the next section) that you would only want to apply during training. Not all data preprocessing needs to be consistent between training and inference.

- Put the preprocessing in functions that are called from both the training and inference pipelines.
- Incorporate the preprocessing into the model itself.
- Use `tf.transform` to create and reuse artifacts.

Let's look at each of these methods. In each of these cases, we'll want to refactor the training pipeline so as to make it easier to reuse all the preprocessing code during inference. The easier it is to reuse code between training and inference, the more likely it is that subtle differences won't crop up and cause training-serving skew.

Reusing Functions

The training pipeline in our case reads TensorFlow Records consisting of already decoded and scaled JPEG files, whereas the prediction pipeline needs to key off the path to an *individual* image file. So, the preprocessing code will not be identical, but we can still collect all the preprocessing into functions that are reused and put them in a class that we'll call `_Preprocessor`.[2] The full code is available in *06b_reuse_functions.ipynb* on GitHub.

The methods of the preprocessor class will be called from two functions, one to create a dataset from TensorFlow Records and the other to create an individual image from a JPEG file. The function to create a preprocessed dataset is:

```
def create_preproc_dataset(pattern):
    preproc = _Preprocessor()
    trainds = tf.data.TFRecordDataset(
        [filename for filename in tf.io.gfile.glob(pattern)]
    ).map(preproc.read_from_tfr).map(
        lambda img, label: (preproc.preprocess(img), label))
    return trainds
```

There are three functions of the preprocessor that are being invoked: the constructor, a way to read TensorFlow Records into an image, and a way to preprocess the image. The function to create an individual preprocessed image is:

```
def create_preproc_image(filename):
    preproc = _Preprocessor()
    img = preproc.read_from_jpegfile(filename)
    return preproc.preprocess(img)
```

2 Ideally, all the functions in this class would be private and only the functions `create_preproc_dataset()` and `create_preproc_image()` would be public. Unfortunately, at the time of writing, `tf.data`'s map functionality doesn't handle the name wrangling that would be needed to use private methods as lambdas. The underscore in the name of the class reminds us that its methods are meant to be private.

Here too, we are using the constructor and preprocessing method, but we're using a different way to read the data. Therefore, the preprocessor will require four methods.

The constructor in Python consists of a method called `__init__()`:

```
class _Preprocessor:
    def __init__(self):
        self.preproc_layers = tf.keras.Sequential([
            tf.keras.layers.experimental.preprocessing.CenterCrop(
                height=IMG_HEIGHT, width=IMG_WIDTH),
                input_shape=(2*IMG_HEIGHT, 2*IMG_WIDTH, 3)
        ])
```

In the `__init__()` method, we set up the preprocessing layers.

To read from a TFRecord we use the `parse_tfr()` function from Chapter 5, now a method of our class:

```
def read_from_tfr(self, proto):
    feature_description = ... # schema
    rec = tf.io.parse_single_example(
        proto, feature_description
    )
    shape = tf.sparse.to_dense(rec['shape'])
    img = tf.reshape(tf.sparse.to_dense(rec['image']), shape)
    label_int = rec['label_int']
    return img, label_int
```

Preprocessing consists of taking the image, sizing it consistently, putting into a batch, invoking the preprocessing layers, and unbatching the result:

```
def preprocess(self, img):
    x = tf.image.resize_with_pad(img, 2*IMG_HEIGHT, 2*IMG_WIDTH)
    # add to a batch, call preproc, remove from batch
    x = tf.expand_dims(x, 0)
    x = self.preproc_layers(x)
    x = tf.squeeze(x, 0)
    return x
```

When reading from a JPEG file, we take care to do all the steps that were carried out when the TFRecord files were written out:

```
def read_from_jpegfile(self, filename):
    # same code as in 05_create_dataset/jpeg_to_tfrecord.py
    img = tf.io.read_file(filename)
    img = tf.image.decode_jpeg(img, channels=IMG_CHANNELS)
    img = tf.image.convert_image_dtype(img, tf.float32)
    return img
```

Now, the training pipeline can create the training and validation datasets using the `create_preproc_dataset()` function that we have defined:

```
train_dataset = create_preproc_dataset(
    'gs://practical-ml-vision-book/flowers_tfr/train-*'
).batch(batch_size)
```

The prediction code (which will go into a serving function, covered in Chapter 9) will take advantage of the `create_preproc_image()` function to read individual JPEG files and then invoke `model.predict()`.

Preprocessing Within the Model

Note that we did not have to do anything special to reuse the model itself for prediction. For example, we did not have to write different variations of the layers: the Hub layer representing MobileNet and the dense layer were all transparently reusable between training and prediction.

Any preprocessing code that we put into the Keras model will be automatically applied during prediction. Therefore, let's take the center-cropping functionality out of the `_Preprocessor` class and move it into the model itself (see *06b_reuse_functions.ipynb* on GitHub for the code):

```
class _Preprocessor:
    def __init__(self):
        # nothing to initialize
        pass

    def read_from_tfr(self, proto):
        # same as before

    def read_from_jpegfile(self, filename):
        # same as before

    def preprocess(self, img):
        return tf.image.resize_with_pad(img, 2*IMG_HEIGHT, 2*IMG_WIDTH)
```

The `CenterCrop` layer moves into the Keras model, which now becomes:

```
layers = [
    tf.keras.layers.experimental.preprocessing.CenterCrop(
        height=IMG_HEIGHT, width=IMG_WIDTH,
        input_shape=(2*IMG_HEIGHT, 2*IMG_WIDTH, IMG_CHANNELS),
    ),
    hub.KerasLayer(...),
    tf.keras.layers.Dense(...),
    tf.keras.layers.Dense(...)
]
```

Recall that the first layer of a Sequential model is the one that carries the `input_shape` parameter. So, we have removed this parameter from the Hub layer and added it to the `CenterCrop` layer. The input to this layer is twice the desired size of the images, so that's what we specify.

The model now includes the `CenterCrop` layer and its output shape is 224x224, our desired output shape:

```
Model: "flower_classification"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
center_crop (CenterCrop)     (None, 224, 224, 3)       0
_________________________________________________________________
mobilenet_embedding (KerasLa (None, 1280)              2257984
_________________________________________________________________
dense_hidden (Dense)         (None, 16)                20496
_________________________________________________________________
flower_prob (Dense)          (None, 5)                 85
```

Of course, if both the training and prediction pipelines read the same data format, we could get rid of the preprocessor completely.

Where to Do Preprocessing

Could we not have also moved the `resize_with_pad()` functionality into the Keras model to get rid of the `_Preprocessor` class entirely? Not at the time this section was being written—Keras models require batched inputs, and it's much easier to write models where batches contain elements of the same shape. Because our input images have different shapes, we need to resize them to something consistent before feeding them to the Keras model. Ragged tensors, which were experimental at the time of writing, will make this unnecessary.

How do you choose whether to have the center cropping done in TensorFlow as part of the `tf.data` pipeline or as a Keras layer and part of the model? Choosing whether to carry out a particular bit of preprocessing in the `tf.data` pipeline or in a Keras layer comes down to five factors:

Efficiency
: If we will always need to carry out center cropping, it is more efficient to have that be part of the preprocessing pipeline because we will be able to cache the results, either by writing out already cropped images into the TensorFlow Records or by adding `.cache()` to the pipeline. If it's done in the Keras model, this preprocessing will have to be carried out during each training iteration.

Experimentation
: Having the center cropping as a Keras layer is more flexible. As just one example, we can choose to experiment with cropping the images at 50% or 70% for different models. We can even treat the cropping ratio as a model hyperparameter.

Maintainability
: Any code in a Keras layer is automatically reused in training and inference, so using Keras layers is less error-prone.

Flexibility
: We haven't discussed this yet, however, when you have operations that need to be carried out differently in training and inference (for example, the data augmentation methods that we will discuss shortly), it is much easier to have those operations be within a Keras layer.

Acceleration
: Commonly, operations in the `tf.data` pipeline are carried out on the CPU and operations in the model function are carried out on the GPU (device placement can be changed, but this is the usual default). Given this default, having code in the model function is a way to take advantage of acceleration and distribution strategies.

Decide where to carry out preprocessing by balancing these considerations. Normally, you will lay out your preprocessing operations and draw a line of separation, doing some operations in `tf.data` and some in the model function.

Note that if you need to preprocess the labels in any way, it's easier to do this in `tf.data` because a Keras Sequential model does not pass the labels through its layers. If you do need to pass the labels through, you will have to switch to the Keras Functional API and pass in a dictionary of features, replacing the image component at each step. See the GAN example in Chapter 12 for an illustration of this approach.

Using tf.transform

What we did in the previous section—writing a `_Preprocessor` class and expecting to keep `read_from_tfr()` and `read_from_jpegfile()` consistent in terms of the preprocessing that is carried out—is hard to enforce. This will be a perennial source of bugs in your ML pipelines because ML engineering teams tend to keep fiddling around with preprocessing and data cleanup routines.

For example, suppose we write out already cropped images into TFRecords for efficiency. How can we ensure that this cropping happens during inference? To mitigate training-serving skew, it is best if we save all the preprocessing operations in an artifacts registry and automatically apply these operations as part of the serving pipeline.

The TensorFlow library that does this is TensorFlow Transform (`tf.transform`). To use `tf.transform`, we need to:

- Write an Apache Beam pipeline to carry out analysis of the training data, precompute any statistics needed for the preprocessing (e.g., mean/variance to use for normalization), and apply the preprocessing.
- Change the training code to read the preprocessed files.
- Change the training code to save the transform function along with the model.

- Change the inference code to apply the saved transform function.

Let's look at each of these briefly (the full code is available in *06h_tftransform.ipynb* on GitHub).

Writing the Beam pipeline

The Beam pipeline to carry out the preprocessing is similar to the pipeline we used in Chapter 5 to convert the JPEG files into TensorFlow Records. The difference is that we use the built-in functionality of TensorFlow Extended (TFX) to create a CSV reader:

```
RAW_DATA_SCHEMA = schema_utils.schema_from_feature_spec({
    'filename': tf.io.FixedLenFeature([], tf.string),
    'label': tf.io.FixedLenFeature([], tf.string),
})
csv_tfxio = tfxio.CsvTFXIO(file_pattern='gs://.../all_data.csv'],
                           column_names=['filename', 'label'],
                           schema=RAW_DATA_SCHEMA)
And we use this class to read the CSV file:
img_records = (p
               | 'read_csv' >> csv_tfxio.BeamSource(batch_size=1)
               | 'img_record' >> beam.Map(
                   lambda x: create_input_record(x[0], x[1]))
              )
```

The input record at this point contains the JPEG data read, and a label index, so we specify this as the schema (see *jpeg_to_tfrecord_tft.py*) to create the dataset that will be transformed:

```
IMG_BYTES_METADATA = tft.tf_metadata.dataset_metadata.DatasetMetadata(
    schema_utils.schema_from_feature_spec({
        'img_bytes': tf.io.FixedLenFeature([], tf.string),
        'label': tf.io.FixedLenFeature([], tf.string),
        'label_int': tf.io.FixedLenFeature([], tf.int64)
    })
)
```

Transforming the data

To transform the data, we pass in the original data and metadata to a function that we call `tft_preprocess()`:

```
raw_dataset = (img_records, IMG_BYTES_METADATA)
transformed_dataset, transform_fn = (
    raw_dataset | 'tft_img' >>
    tft_beam.AnalyzeAndTransformDataset(tft_preprocess)
)
```

The preprocessing function carries out the resizing operations using TensorFlow functions:

```
def tft_preprocess(img_record):
    img = tf.map_fn(decode_image, img_record['img_bytes'],
                    fn_output_signature=tf.uint8)
    img = tf.image.convert_image_dtype(img, tf.float32)
    img = tf.image.resize_with_pad(img, IMG_HEIGHT, IMG_WIDTH)
    return {
        'image': img,
        'label': img_record['label'],
        'label_int': img_record['label_int']
    }
```

Saving the transform

The resulting transformed data is written out as before. In addition, the transformation function is written out:

```
transform_fn | 'write_tft' >> tft_beam.WriteTransformFn(
    os.path.join(OUTPUT_DIR, 'tft'))
```

This creates a SavedModel that contains all the preprocessing operations that were carried out on the raw dataset.

Reading the preprocessed data

During training, the transformed records can be read as follows:

```
def create_dataset(pattern, batch_size):
    return tf.data.experimental.make_batched_features_dataset(
        pattern,
        batch_size=batch_size,
        features = {
            'image': tf.io.FixedLenFeature(
                [IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS], tf.float32),
            'label': tf.io.FixedLenFeature([], tf.string),
            'label_int': tf.io.FixedLenFeature([], tf.int64)
        }
    ).map(
        lambda x: (x['image'], x['label_int'])
    )
```

These images are already scaled and resized and so can be used directly in the training code.

Transformation during serving

We need to make the transformation function artifacts (that were saved using `WriteTransformFn()`) available to the prediction system. We can do this by ensuring that the `WriteTransformFn()` writes the transform artifacts to a Cloud Storage location that is accessible to the serving system. Alternatively, the training pipeline can copy over the transform artifacts so that they are available alongside the exported model.

At prediction time, all the scaling and preprocessing operations are loaded and applied to the image bytes sent from the client:

```
preproc = tf.keras.models.load_model(
    '.../tft/transform_fn').signatures['transform_signature']
preprocessed = preproc(img_bytes=tf.convert_to_tensor(img_bytes)...)
```

We then call `model.predict()` on the preprocessed data:

```
pred_label_index = tf.math.argmax(model.predict(preprocessed))
```

In Chapter 7, we will look at how to write a serving function that does these operations on behalf of the client.

Benefits of tf.transform

Note that with `tf.transform` we have avoided having to make the trade-offs inherent in either putting preprocessing code in the `tf.data` pipeline or including it as part of the model. We now get the best of both approaches—efficient training and transparent reuse to prevent training-serving skew:

- The preprocessing (scaling and resizing of input images) happens only once.
- The training pipeline reads already preprocessed images, and is therefore fast.
- The preprocessing functions are stored into a model artifact.
- The serving function can load the model artifact and apply the preprocessing before invoking the model (details on how are covered shortly).

The serving function doesn't need to know the details of the transformations, only where the transform artifacts are stored. A common practice is to copy over these artifacts to the model output directory as part of the training program, so that they are available alongside the model itself. If we change the preprocessing code, we simply run the preprocessing pipeline again; the model artifact containing the preprocessing code gets updated, so the correct preprocessing gets applied automatically.

There are other advantages to using `tf.transform` beyond preventing training-serving skew. For example, because `tf.transform` iterates over the entire dataset once before training has even started, it is possible to use global statistics of the dataset (e.g., the mean) to scale the values.

Data Augmentation

Preprocessing is useful for more than simply reformatting images to the size and shape required by the model. Preprocessing can also be a way to improve model quality through *data augmentation*.

Data augmentation is a data-space solution to the problem of insufficient data (or insufficient data of the right kind)—it is a set of techniques that enhance the size and quality of training datasets with the goal of creating machine learning models that are more accurate and that generalize better.

Deep learning models have lots of weights, and the more weights there are, the more data is needed to train the model. If our dataset is too small relative to the size of the ML model, the model can employ its parameters to memorize the input data, which results in overfitting (a condition where the model performs well on training data, but produces poor results on unseen data at inference time).

As a thought experiment, consider an ML model with one million weights. If we have only 10,000 training images, the model can assign 100 weights to each image, and these weights can home in on some characteristic of each image that makes it unique in some way—for example, perhaps this is the only image where there is a bright patch centered around a specific pixel. The problem is that such an overfit model will not perform well after it is put into production. The images that the model will be required to predict will be different from the training images, and the noisy information it has learned won't be helpful. We need the ML model to *generalize* from the training dataset. For that to happen, we need a lot of data, and the larger the model we want, the more data we need.

Data augmentation techniques involve taking the images in the training dataset and transforming them to create new training examples. Existing data augmentation methods fall into three categories:

- Spatial transformation, such as random zooming, cropping, flipping, rotation, and so on
- Color distortion to change brightness, hue, etc.
- Information dropping, such as random masking or erasing of different parts of the image

Let's look at each of these in turn.

Spatial Transformations

In many cases, we can flip or rotate an image without changing its essence. For example, if we are trying to detect types of farm equipment, flipping the images horizontally (left to right, as shown in the top row of Figure 6-9) would simply simulate the equipment as seen from the other side. By augmenting the dataset using such image transformations, we are providing the model with more variety—meaning more examples of the desired image object or class in varying sizes, spatial locations, orientations, etc. This will help create a more robust model that can handle these kinds of variations in real data.

Figure 6-9. Some geometric transformations of an image of a tractor in a field. Photograph by author.

However, flipping the image vertically (top to bottom, as shown on the left side of Figure 6-9) is not a good idea, for a few reasons. First, the model is not expected to correctly classify an upside-down image in production, so there is no point in adding this image to the training dataset. Second, a vertically flipped tractor image makes it more difficult for the ML model to identify features like the cabin that are not vertically symmetric. Flipping the image vertically thus both adds an image type that the model is not required to classify correctly and makes the learning problem tougher.

Make sure that augmenting data makes the training dataset larger, but does not make the problem more difficult. In general, this is the case only if the augmented image is typical of the images that the model is expected to predict on, and not if the augmentation creates a skewed, unnatural image. Information dropping methods, discussed shortly, are an exception to this rule.

Keras supports several data augmentation layers, including `RandomTranslation`, `RandomRotation`, `RandomZoom`, `RandomCrop`, `RandomFlip`, and so on. They all work similarly.

The `RandomFlip` layer will, during training, randomly either flip an image or keep it in its original orientation. During inference, the image is passed through unchanged.

Keras does this automatically; all we have to do is add this as one of the layers in our model:

```
tf.keras.layers.experimental.preprocessing.RandomFlip(
    mode='horizontal',
    name='random_lr_flip/none'
)
```

The `mode` parameter controls the types of flips that are allowed, with a `horizontal` flip being the one that flips the image left to right. Other modes are `vertical` and `horizontal_and_vertical`.

In the previous section, we center cropped the images. When we do a center crop, we lose a considerable part of the image. To improve our training performance, we could consider augmenting the data by taking random crops of the desired size from the input images. The `RandomCrop` layer in Keras will do random crops during training (so that the model sees different parts of each image during each epoch, although some of them will now include the padded edges and may not even include the parts of the image that are of interest) and behave like a `CenterCrop` during inference.

The full code for this example is in *06d_augmentation.ipynb* on GitHub. Combining these two operations, our model layers now become:

```
layers = [
    tf.keras.layers.experimental.preprocessing.RandomCrop(
        height=IMG_HEIGHT//2, width=IMG_WIDTH//2,
        input_shape=(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS),
        name='random/center_crop'
    ),
    tf.keras.layers.experimental.preprocessing.RandomFlip(
        mode='horizontal',
        name='random_lr_flip/none'
    ),
    hub.KerasLayer(
        "https://tfhub.dev/.../mobilenet_v2/...",
        trainable=False,
        name='mobilenet_embedding'),
    tf.keras.layers.Dense(
        num_hidden,
        kernel_regularizer=regularizer,
        activation=tf.keras.activations.relu,
        name='dense_hidden'),
    tf.keras.layers.Dense(
        len(CLASS_NAMES),
        kernel_regularizer=regularizer,
        activation='softmax',
        name='flower_prob')
]
```

And the model itself becomes:

```
Model: "flower_classification"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
random/center_crop (RandomCr (None, 224, 224, 3)       0
_________________________________________________________________
random_lr_flip/none (RandomF (None, 224, 224, 3)       0
_________________________________________________________________
mobilenet_embedding (KerasLa (None, 1280)              2257984
_________________________________________________________________
dense_hidden (Dense)         (None, 16)                20496
_________________________________________________________________
flower_prob (Dense)          (None, 5)                 85
```

Training this model is similar to training without augmentation. However, we will need to train the model longer whenever we augment data—intuitively, we need to train for twice as many epochs in order for the model to see both flips of the image. The result is shown in Figure 6-10.

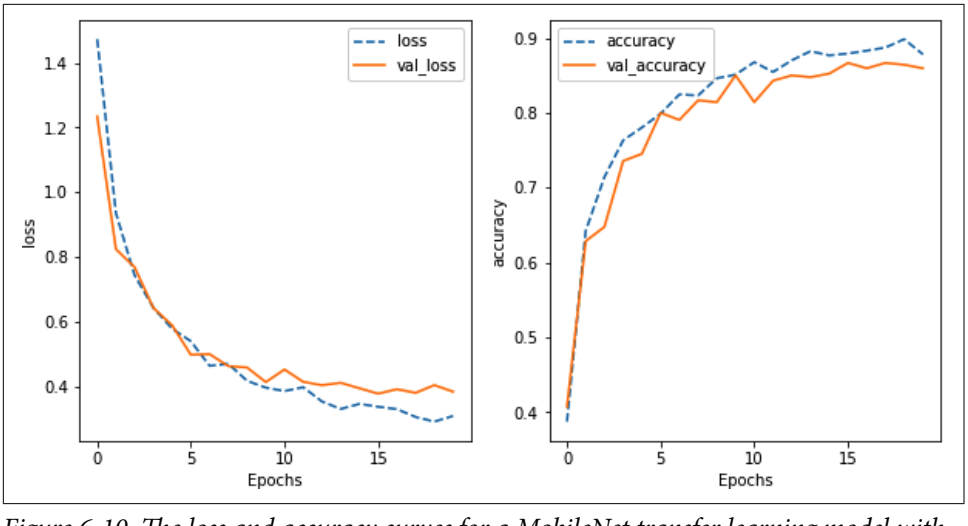


Figure 6-10. The loss and accuracy curves for a MobileNet transfer learning model with data augmentation. Compare to Figure 6-7.

Comparing Figure 6-10 with Figure 6-7, we notice how much more resilient the model training has become with the addition of data augmentation. Note that the training and validation loss are pretty much in sync, as are the training and validation accuracies. The accuracy, at 0.86, is only slightly better than before (0.85); the important thing is that we can be more confident about this accuracy because of the much better behaved training curves.

By adding data augmentation, we have dramatically lowered the extent of overfitting.

Color Distortion

It's important to not limit yourself to the set of augmentation layers that are readily available. Think instead about what kinds of variations of the images the model is likely to encounter in production. For example, it is likely that photographs provided to an ML model (especially if these are photographs by amateur photographers) will vary quite considerably in terms of lighting. We can therefore increase the effective size of the training dataset and make the ML model more resilient if we augment the data by randomly changing the brightness, contrast, saturation, etc. of the training images. While Keras has several built-in data augmentation layers (like `RandomFlip`), it doesn't currently support changing the contrast[3] and brightness. So, let's implement this ourselves.

We'll create a data augmentation layer from scratch that will randomly change the contrast and brightness of an image. The class will inherit from the Keras `Layer` class and take two arguments, the ranges within which to adjust the contrast and the brightness (the full code is in *06e_colordistortion.ipynb* on GitHub):

```
class RandomColorDistortion(tf.keras.layers.Layer):
    def __init__(self, contrast_range=[0.5, 1.5],
                 brightness_delta=[-0.2, 0.2], **kwargs):
        super(RandomColorDistortion, self).__init__(**kwargs)
        self.contrast_range = contrast_range
        self.brightness_delta = brightness_delta
```

When invoked, this layer will need to behave differently depending on whether it is in training mode or not. If not in training mode, the layer will simply return the original images. If it is in training mode, it will generate two random numbers, one to adjust the contrast within the image and the other to adjust the brightness. The actual adjustment is carried out using methods available in the `tf.image` module:

```
    def call(self, images, training=False):
        if not training:
            return images

        contrast = np.random.uniform(
            self.contrast_range[0], self.contrast_range[1])
        brightness = np.random.uniform(
            self.brightness_delta[0], self.brightness_delta[1])

        images = tf.image.adjust_contrast(images, contrast)
        images = tf.image.adjust_brightness(images, brightness)
        images = tf.clip_by_value(images, 0, 1)
        return images
```

3 `RandomContrast` was added between the time this section was written and when the book went to press.

It's important that the implementation of the custom augmentation layer consists of TensorFlow functions so that these functions can be implemented efficiently on a GPU. See Chapter 7 for recommendations on writing efficient data pipelines.

The effect of this layer on a few training images is shown in Figure 6-11. Note that the images have different contrast and brightness levels. By invoking this layer many times on each input image (once per epoch), we ensure that the model gets to see many color variations of the original training images.

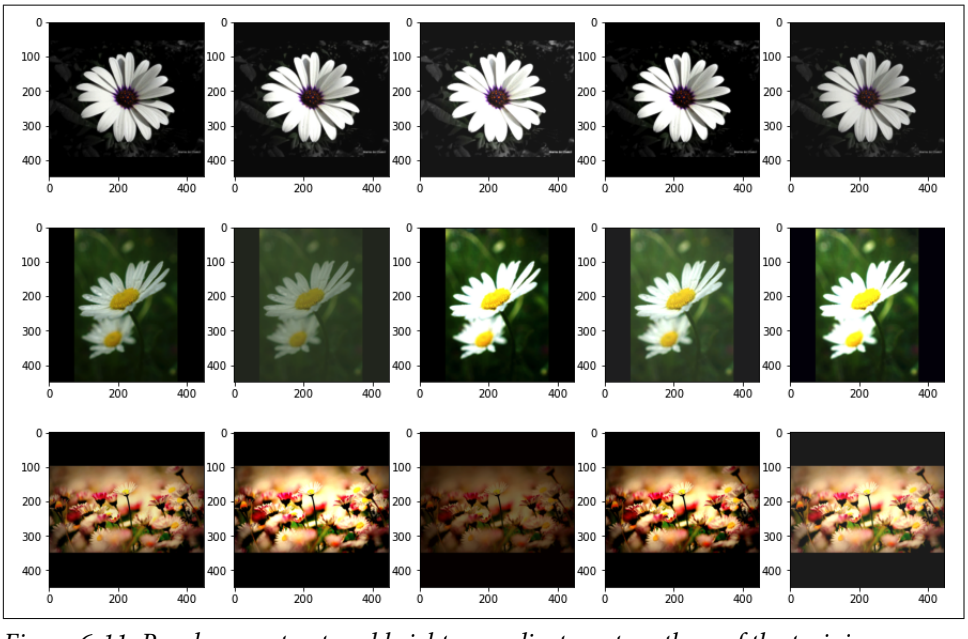


Figure 6-11. Random contrast and brightness adjustment on three of the training images. The original images are shown in the first panel of each row, and four generated images are shown in the other panels. If you're looking at grayscale images, please refer to 06e_colordistortion.ipynb on GitHub to see the effect of the color distortion.

The layer itself can be inserted into the model after the `RandomFlip` layer:

```
layers = [
    ...
    tf.keras.layers.experimental.preprocessing.RandomFlip(
        mode='horizontal',
        name='random_lr_flip/none'
    ),
    RandomColorDistortion(name='random_contrast_brightness/none'),
    hub.KerasLayer ...
]
```

The full model will then have this structure:

```
Model: "flower_classification"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
random/center_crop (RandomCr (None, 224, 224, 3)       0
_________________________________________________________________
random_lr_flip/none (RandomF (None, 224, 224, 3)       0
_________________________________________________________________
random_contrast_brightness/n (None, 224, 224, 3)       0
_________________________________________________________________
mobilenet_embedding (KerasLa (None, 1280)              2257984
_________________________________________________________________
dense_hidden (Dense)         (None, 16)                20496
_________________________________________________________________
flower_prob (Dense)          (None, 5)                 85
=================================================================
Total params: 2,278,565
Trainable params: 20,581
Non-trainable params: 2,257,984
```

Training of the model remains identical. The result is shown in Figure 6-12. We get better accuracy than with just geometric augmentation (0.88 instead of 0.86) and the training and validation curves remain totally in sync, indicating that overfitting is under control.

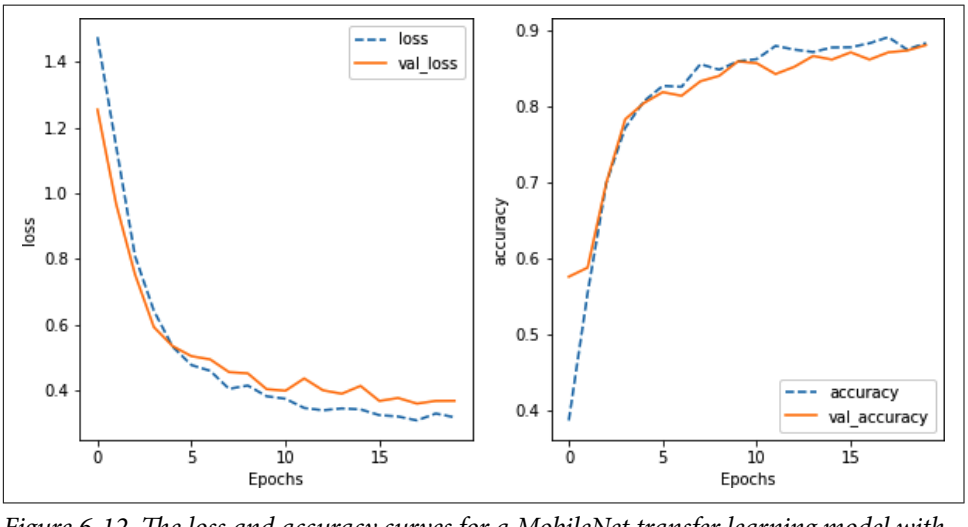


Figure 6-12. The loss and accuracy curves for a MobileNet transfer learning model with geometric and color augmentation. Compare to Figure 6-7 and 6-10.

Information Dropping

Recent research highlights some new ideas in data augmentation that involve making more dramatic changes to the images. These techniques *drop* information from the images in order to make the training process more resilient and to help the model attend to the important features of the images. They include:

Cutout
: Randomly mask out square regions of input during training. This helps the model learn to disregard uninformative parts of the image (such as the sky) and attend to the discriminative parts (such as the petals).

Mixup
: Linearly interpolate a pair of training images and assign as their label the corresponding interpolated label value.

CutMix
: A combination of cutout and mixup. Cut patches from different training images and mix the ground truth labels proportionally to the area of the patches.

GridMask
: Delete uniformly distributed square regions while controlling the density and size of the deleted regions. The underlying assumption is that images are intentionally collected—uniformly distributed square regions tend to be the background.

Cutout and GridMask involve preprocessing operations on a single image and can be implemented similar to how we implemented the color distortion. Open source code for cutout and GridMask is available on GitHub.

Mixup and CutMix, however, use information from multiple training images to create synthetic images that may bear no resemblance to reality. In this section we'll look at how to implement mixup, since it is simpler. The full code is in *06f_mixup.ipynb* on GitHub.

The idea behind mixup is to linearly interpolate a pair of training images and their labels. We can't do this in a Keras custom layer because the layer only receives images; it doesn't get the labels. Therefore, let's implement a function that receives a batch of images and labels and does the mixup:

```
def augment_mixup(img, label):
    # parameters
    fracn = np.rint(MIXUP_FRAC * len(img)).astype(np.int32)
    wt = np.random.uniform(0.5, 0.8)
```

In this code, we have defined two parameters: `fracn` and `wt`. Instead of mixing up all the images in the batch, we will mix up a fraction of them (by default, 0.4) and keep the remaining images (and labels) as they are. The parameter `fracn` is the number of