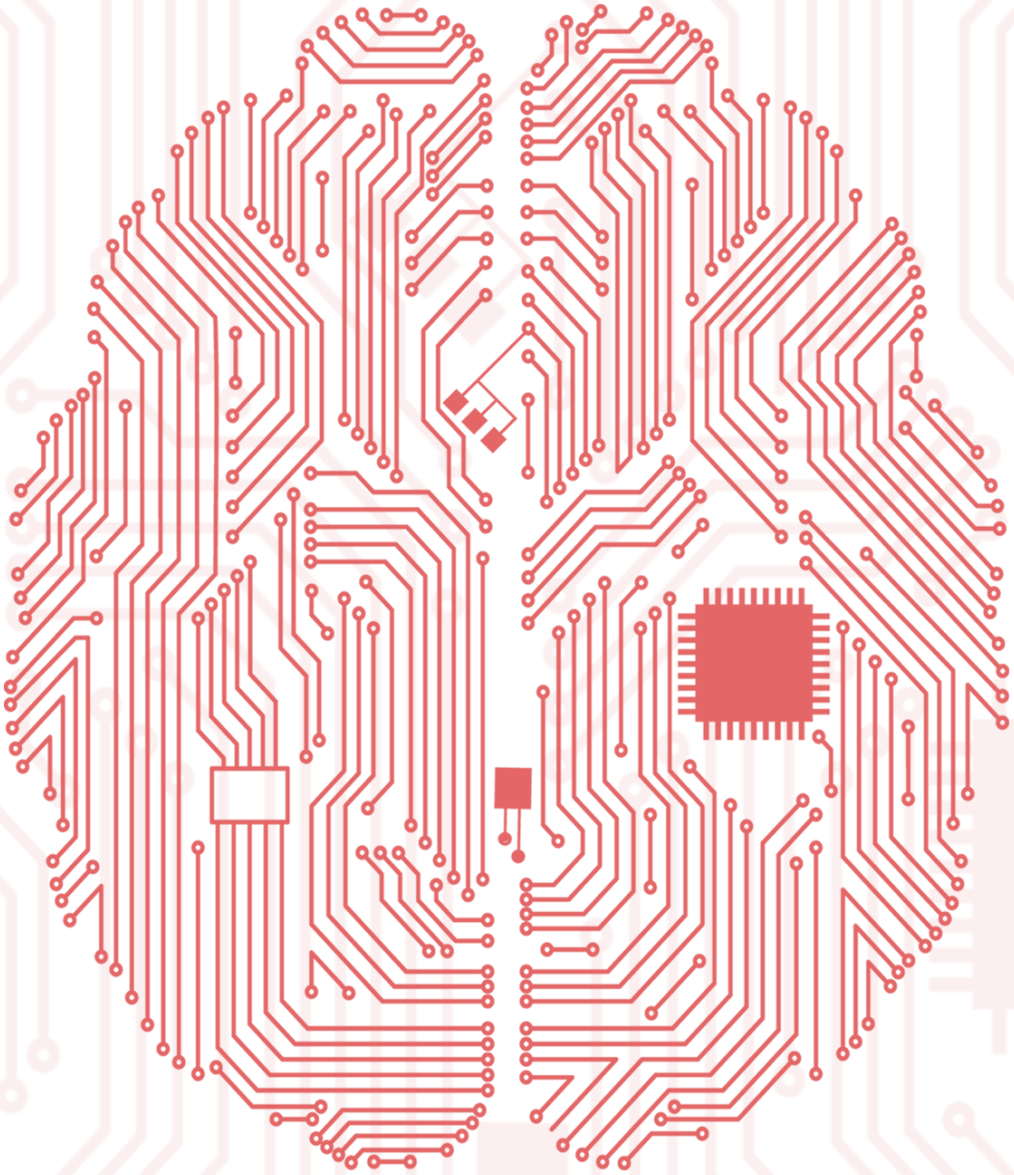
DEEP LEARNING FOR COMPUTER VISION



WITH PYTHON

Dr. Adrian Rosebrock

pyimagesearch

Deep Learning for Computer Vision with Python

Starter Bundle

Dr. Adrian Rosebrock

1st Edition (1.1.0)

Copyright © 2017 Adrian Rosebrock, PyImageSearch.com

PUBLISHED BY PYIMAGESEARCH

PYIMAGESEARCH.COM

The contents of this book, unless otherwise indicated, are Copyright ©2017 Adrian Rosebrock, PyimageSearch.com. All rights reserved. Books like this are made possible by the time invested by the authors. If you received this book and did not purchase it, please consider making future books possible by buying a copy at https://www.pyimagesearch.com/deep-learning-computer-vision-python-book/ today.

First printing, September 2017

To my father, Joe; my wife, Trisha;
and the family beagles, Josie and Jemma.
Without their constant love and support,
this book would not be possible.

Contents

Companion Website

Thank you for picking up a copy of *Deep Learning for Computer Vision with Python!* To accompany this book I have created a companion website which includes:

- Up-to-date installation instructions on how to configure your development environment
- Instructions on how to use the pre-configured Ubuntu VirtualBox virtual machine and Amazon Machine Image (AMI)
- Supplementary material that I could not fit inside this book

Additionally, you can use the "Issues" feature inside the companion website to report any bugs, typos, or problems you encounter when working through the book. I don't expect many problems; however, this is a brand new book so myself and other readers would appreciate reporting any issues you run into. From there, I can keep the book updated and bug free.

To create your companion website account, just use this link:

http://pyimg.co/fnkxk

Take a second to create your account now so you'll have access to the supplementary materials as you work through the book.

1. Introduction

"The secret of getting ahead is to get started." – Mark Twain

Welcome to *Deep Learning for Computer Vision with Python*. This book is your guide to mastering deep learning applied to practical, real-world computer vision problems utilizing the Python programming language and the Keras + mxnet libraries. Inside this book, you'll learn how to apply deep learning to take-on projects such as image classification, object detection, training networks on large-scale datasets, and *much more*.

Deep Learning for Computer Vision with Python strives to be the *perfect balance* between ***theory taught in a classroom/textbook and the actual hands-on knowledge you'll need to be successful in the real world***.

To accomplish this goal, you'll learn in a *practical, applied* manner by training networks on your own custom datasets and even competing in challenging state-of-the-art image classification challenges and competitions. By the time you finish this book, you'll be *well equipped* to apply deep learning to your own projects. And with enough practice, I have no doubt that you'll be able to leverage your newly gained knowledge to find a job in the deep learning space, become a deep learning for computer vision consultant/contractor, or even start your own computer vision-based company that leverages deep learning.

So grab your highlighter. Find a comfortable spot. And let me help you on your journey to deep learning mastery. Remember the most important step is the first one – to simply get started.

1.1 I Studied Deep Learning the *Wrong Way*... This Is the *Right Way*

I want to start this book by sharing a personal story with you:

Toward the end of my graduate school career (2013-2014), I started wrapping my head around this whole "deep learning" thing due to a timing quirk. I was in a very unique situation. My dissertation was (essentially) wrapped up. Each of my Ph.D. committee members had signed off on it. However, due to university/department regulations, I still had an extra semester that I needed to "hang around" for before I could officially defend my dissertation and graduate. This gap essentially

left me with an entire semester ($\approx$ 4 months) to kill – **it was an *excellent* time to start studying deep learning**.

My first stop, as is true for most academics, was to read through all the recent publications on deep learning. Due to my machine learning background, it didn't take long to grasp the actual *theoretical foundations* of deep learning.

However, I'm of the opinion that until you take your *theoretical knowledge* and *implement it*, you haven't actually *learned* anything yet. Transforming *theory* into *implementation* is a **very** different process, as any computer scientist who has taken a data structures class before will tell you: *reading about* red-black trees and then actually *implementing them from scratch* requires two different skill sets.

And that's *exactly* what my problem was.

After reading these deep learning publications, I was left scratching my head; I couldn't take what I learned from the papers and *implement* the actual algorithms, let alone *reproduce* the results.

Frustrated with my failed attempts at implementation, I spent *hours* searching on Google, hunting for deep learning tutorials, only to come up empty-handed. Back then, there weren't many deep learning tutorials to be found.

Finally, I resorted to playing around with libraries and tools such as Caffe, Theano, and Torch, blindly followed poorly written blog posts (with mixed results, to say the least).

I *wanted* to get started, but nothing had actually *clicked* yet – the deep learning lightbulb in my head was stuck in the "off" position.

To be totally honest with you, it was a painful, emotionally trying semester. I could *clearly* see the value of deep learning for computer vision, but I had nothing to show for my effort, except for a stack of deep learning papers on my desk that I *understood* but struggled to *implement*.

During the last month of the semester, I finally found my way to deep learning success through hundreds of trial-and-error experiments, countless late nights, and *a lot* of perseverance. In the long run, those four months made a massive impact on my life, my research path, and how I understand deep learning today...

...but I would *not* advise you to take the same path I did.

If you take *anything* from my personal experience, it should be this:

1. You don't need a decade of theory to get started in deep learning.
2. You don't need pages and pages of equations.
3. And you certainly don't need a degree in computer science (although it can be helpful).

When I got started studying deep learning, I made the critical mistake of taking a deep dive into the publications without ever resurfacing to try and implement what I studied. Don't get me wrong – theory *is* important. But if you don't (or can't) take your newly minted theoretical knowledge and use apply it to build actual real-world applications, you'll struggle to find your space in the deep learning world.

Deep learning, and most other higher-level, specialized computer science subjects are recognizing that *theoretical knowledge is not enough* – **we need to be *practitioners* in our respective fields as well**. In fact, the concept of becoming a deep learning *practitioner* was my ***exact motivation*** in writing *Deep Learning for Computer Vision with Python*.

While there are:

1. Textbooks that will teach you the theoretical underpinnings of machine learning, neural networks, and deep learning
2. And countless "cookbook"-style resources that will "show you in code", but never relate the code back to true theoretical knowledge...

...*none* of these books or resources will serve as the bridge between the other.

On one side of the bridge you have your textbooks, deeply rooted in theory and abstraction. And on the other side, you have "show me in code" books that simply present examples to you,

perhaps explaining the code, but never relating the code back to the underlying theory.

There is a fundamental disconnect between these two styles of learning, a gap that I want to help fill so you can learn in a *better, more efficient* way.

I thought back to my graduate school days, to my feelings of frustration and irritation, to the days when I even considered giving up. I channeled these feelings as I sat down to write this book. ***The book you're reading now is the book I wish I had when I first started studying deep learning.***

Inside the remainder of *Deep Learning for Computer Vision with Python*, you'll find **super practical walkthroughs**, **hands-on tutorials (with lots of code)**, and a **no-nonsense teaching style** that is guaranteed to cut through all the cruft and help you master deep learning for computer vision.

Rest assured, you're in good hands – this is the *exact* book that you've been looking for, and I'm incredibly excited to be joining you on your deep learning for visual recognition journey.

1.2 Who This Book Is For

This book is for **developers**, **researchers**, and **students** who want to become proficient in deep learning for computer vision and visual recognition.

1.2.1 Just Getting Started in Deep Learning?

Don't worry. You won't get bogged down by tons of theory and complex equations. We'll start off with the basics of machine learning and neural networks. You'll learn in a fun, practical way with lots of code. I'll also provide you with references to seminal papers in the machine learning literature that you can use to extend your knowledge once you feel like you have a solid foundation to stand on.

The most important step you can take right now is to simply *get started*. Let me take care of the teaching – regardless of your skill level, trust me, you will not get left behind. By the time you finish the first few chapters of this book, you'll be a neural network ninja and be able to graduate to the more advanced content.

1.2.2 Already a Seasoned Deep Learning Practitioner?

This book isn't just for beginners – *there's advanced content in here too*. For each chapter in this book, I provide a set of academic references you can use to further your knowledge. Many chapters inside *Deep Learning for Computer Vision with Python* actually *explain* these academic concepts in a manner that is easily understood and digested.

Best of all, the solutions and tactics I provide inside this book can be directly applied to your current job and research. The time you'll save by reading through *Deep Learning for Computer Vision with Python* will *more than pay for itself* once you apply your knowledge to your projects/research.

1.3 Book Organization

Since this book covers a *huge* amount of content, I've broken down the book into **three volumes** called ***"bundles"***. Each bundle sequentially builds on top of the other and includes all chapters from the lower volumes. You can find a quick breakdown of the bundles below.

1.3.1 Volume #1: Starter Bundle

The Starter Bundle is a great fit if you're taking your first steps toward deep learning for image classification mastery.

You'll learn the basics of:

1. Machine learning
2. Neural Networks
3. Convolutional Neural Networks
4. How to work with your own custom datasets

1.3.2 Volume #2: Practitioner Bundle

The Practitioner Bundle builds on the Starter Bundle and is perfect if you want to study deep learning *in-depth*, understand advanced techniques, and discover common best practices and rules of thumb.

1.3.3 Volume #3: ImageNet Bundle

The ImageNet Bundle is the *complete deep learning for computer vision experience*. In this volume of the book, I demonstrate how to train large-scale neural networks on the *massive* ImageNet dataset as well as tackle real-world case studies, including age + gender prediction, vehicle make + model identification, facial expression recognition, *and much more*.

1.3.4 Need to Upgrade Your Bundle?

If you would ever like to upgrade your bundle, all you have to do is send me a message and we can get the upgrade taken care of ASAP:

http://www.pyimagesearch.com/contact/

1.4 Tools of the Trade: Python, Keras, and Mxnet

We'll be utilizing the *Python* programming language for all examples in this book. Python is an extremely easy language to learn. It has intuitive syntax. Is super powerful. And it's **the best way** to work with deep learning algorithms.

The primary deep learning library we'll be using is Keras [1]. The Keras library is maintained by the brilliant François Chollet, a deep learning researcher and engineer at Google. I have been using Keras for *years* and can say that it's hands-down my favorite deep learning package. As a minimal, modular network library that can use *either* Theano or TensorFlow as a backend, you just can't beat Keras.

The second deep learning library we'll be using is mxnet [2] (ImageNet Bundle only), a lightweight, portable, and flexible deep learning library. The mxnet package provides bindings to the Python programming language and specializes in *distributed, multi-machine learning* – the ability to parallelize training across GPUs/devices/nodes is *critical* when training deep neural network architectures on massive datasets (such as ImageNet).

Finally, we'll also be using a few computer vision, image processing, and machine learning libraries such as OpenCV, scikit-image, scikit-learn, etc.

Python, Keras, and mxnet are well-built tools that when combined tighter create a *powerful deep learning development environment* that you can use to master deep learning for visual recognition.

1.4.1 What About TensorFlow?

TensorFlow [3] and Theano [4] are libraries for defining abstract, general-purpose computation graphs. While they are used for deep learning, they are *not* deep learning frameworks and are in fact used for a great many other applications than deep learning.

Keras, on the other hand, *is* a deep learning framework that provides a well-designed API to facilitate building deep neural networks with ease. Under the hood, Keras uses *either* the

TensorFlow or Theano computational backend, allowing it to take advantage of these powerful computation engines.

Think of a computational backend as an engine that runs in your car. You can replace parts in your engine, optimize others, or replace the engine entirely (provided the engine adheres to a set of specifications of course). Utilizing Keras gives you *portability* across engines and the ability to choose the best engine for your project.

Thinking of this at a different angle, using TensorFlow or Theano to build a deep neural network would be akin to utilizing strictly NumPy to build a machine learning classifier.

Is it possible? Absolutely.

However, it's more beneficial to use a library that is *dedicated* to machine learning, such as scikit-learn [5], rather than reinvent the wheel with NumPy (and at the expense of an order of magnitude more code).

In the same vein, Keras *sits on top* of TensorFlow or Theano, enjoying:

1. The benefits of a powerful underlying computation engine
2. An API that makes it easier for you to build your own deep learning networks

Furthermore, since Keras will be added to the core TensorFlow library at Google [6], we can always integrate TensorFlow code *directly* into our Keras models if we so wish. In many ways, we are getting the best of both worlds by using Keras.

1.4.2 Do I Need to Know OpenCV?

You *do not* need to know the OpenCV computer vision and image processing library [7] to be successful when going through this book.

We only use OpenCV to facilitate basic image processing operations such as loading an image from disk, displaying it to our screen, and a few other basic operations.

That said, a little bit of OpenCV experience goes a long way, so if you're new to OpenCV and computer vision, I *highly recommend* that you work through this book and my other publication, *Practical Python and OpenCV* [8] in tandem.

Remember, deep learning is only *one facet* of computer vision – there are a number of computer vision techniques you should study to round out your knowledge.

1.5 Developing Our Own Deep Learning Toolset

One of the inspirations behind writing this book was to demonstrate using *existing deep learning libraries* to build our own custom Python-based toolset, enabling us to train our own deep learning networks.

However, this book isn't just *any* deep learning toolkit . . . this toolkit is the *exact same one* I have developed and refined over the past few years doing deep learning research and development of my own.

As we progress through this book, we'll build components of this toolset one at a time. By the end of *Deep Learning for Computer Vision with Python*, our toolset will be able to:

1. Load image datasets from disk, store them in memory, or write them to an optimized database format.
2. Preprocess images such that they are suitable for training a Convolutional Neural Network.
3. Create a blueprint class that can be used to build our own custom implementations of Convolutional Neural Networks.
4. Implement popular CNN architectures by hand, such as AlexNet, VGGNet, GoogLeNet, ResNet, and SqueezeNet (and train them from scratch).
5. . . . *and much more!*

1.6 Summary

We are living in a special time in machine learning, neural network, and deep learning history. Never in the history of machine learning and neural networks have the available tools at our disposable been so exceptional.

From a software perspective, we have libraries such as Keras and mxnet complete with Python bindings, enabling us to rapidly construct deep learning architectures in a *fraction of the time* it took us just years before.

Then, from a hardware view, general purpose GPUs are becoming *increasingly cheaper* while *still becoming more powerful*. The advances in GPU technology alone have allowed anyone with a modest budget to build even a simple gaming rig to conduct *meaningful* deep learning research.

Deep learning is an *exciting field*, and due to these powerful GPUs, modular libraries, and unbridled, intelligent researchers, we're seeing new publications that push the state-of-the-art coming on a *monthly basis*.

You see, *now* is the time to undertake studying deep learning for computer vision.

Don't miss out on this time in history – not only should you be a part of deep learning, but those who capitalize early are sure to see immense returns on their investment of time, resources, and creativity.

Enjoy this book. I'm excited to see where it takes you in this amazing field.

2. What Is Deep Learning?

> *"Deep learning methods are representation-learning methods with multiple levels of representation, obtained by composing simple but nonlinear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. [...] The key aspect of deep learning is that these layers are not designed by human engineers: they are learned from data using a general-purpose learning procedure"* – Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, Nature 2015. [9]

Deep learning is a subfield of machine learning, which is, in turn, a subfield of artificial intelligence (AI). For a graphical depiction of this relationship, please refer to Figure 2.1.

The central goal of AI is to provide a set of algorithms and techniques that can be used to solve problems that humans perform *intuitively* and *near automatically*, but are otherwise very challenging for computers. A great example of such a class of AI problems is interpreting and understanding the contents of an image – this task is something that a human can do with little-to-no effort, but it has proven to be *extremely difficult* for machines to accomplish.

While AI embodies a large, diverse set of work related to automatic machine reasoning (inference, planning, heuristics, etc.), the machine learning subfield tends to be *specifically interested* in **pattern recognition** and **learning from data**.

Artificial Neural Networks (ANNs) are a class of machine learning algorithms that learn from data and specialize in pattern recognition, inspired by the structure and function of the brain. As we'll find out, deep learning belongs to the family of ANN algorithms, and in most cases, the two terms can be used interchangeably. In fact, you may be surprised to learn that the deep learning field has been around for over *60 years*, going by different names and incarnations based on research trends, available hardware and datasets, and popular options of prominent researchers at the time.

In the remainder of this chapter, we'll review a brief history of deep learning, discuss what makes a neural network "deep", and discover the concept of "hierarchical learning" and how it has made deep learning one of the major success stories in modern day machine learning and computer vision.

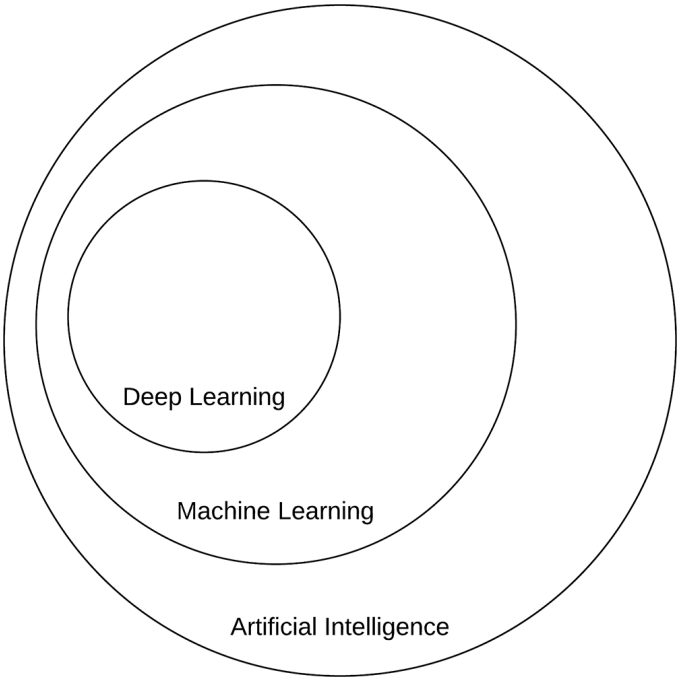


Figure 2.1: A Venn diagram describing deep learning as a subfield of machine learning which is in turn a subfield of artificial intelligence (Image inspired by Figure 1.4 of Goodfellow et al. [10]).

2.1 A Concise History of Neural Networks and Deep Learning

The history of neural networks and deep learning is a long, somewhat confusing one. It may surprise you to know that "deep learning" has existed since the 1940s undergoing various name changes, including *cybernetics*, *connectionism*, and the most familiar, *Artificial Neural Networks* (ANNs).

While *inspired* by the human brain and how its neurons interact with each other, ANNs are *not* meant to be realistic models of the brain. Instead, they are an inspiration, allowing us to draw parallels between a very basic model of the brain and how we can mimic some of this behavior through artificial neural networks. We'll discuss ANNs and the relation to the brain in Chapter 10.

The first neural network model came from McCulloch and Pitts in 1943 [11]. This network was a *binary classifier*, capable of recognizing two different categories based on some input. The problem was that the *weights* used to determine the class label for a given input needed to be *manually tuned* by a human – this type of model clearly does not scale well if a human operator is required to intervene.

Then, in the 1950s the seminal Perceptron algorithm was published by Rosenblatt [12, 13] – this model could *automatically* learn the weights required to classify an input (no human intervention required). An example of the Perceptron architecture can be seen in Figure 2.2. In fact, this automatic training procedure formed the basis of Stochastic Gradient Descent (SGD) which is still used to train *very deep* neural networks today.

During this time period, Perceptron-based techniques were all the rage in the neural network community. However, a 1969 publication by Minsky and Papert [14] effectively stagnated neural network research for nearly a decade. Their work demonstrated that a Perceptron with a linear activation function (regardless of depth) was merely a linear classifier, unable to solve nonlinear problems. The canonical example of a nonlinear problem is the XOR dataset in Figure 2.3. Take a second now to convince yourself that it is *impossible* to try a *single line* that can separate the blue

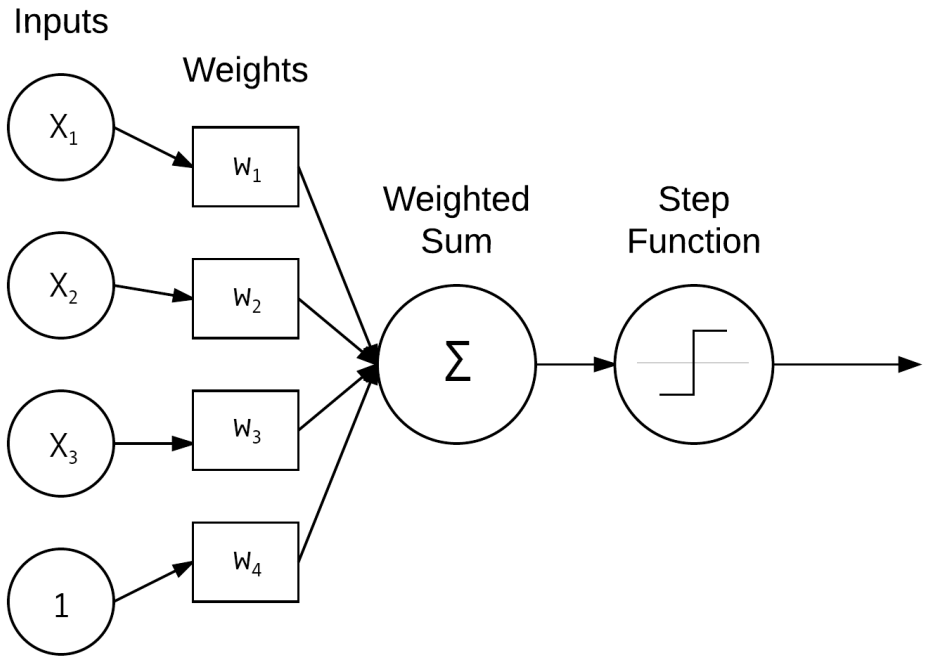


Figure 2.2: An example of the simple Perceptron network architecture that accepts a number of inputs, computes a weighted sum, and applies a step function to obtain the final prediction. We'll review the Perceptron in detail inside Chapter 10.

stars from the red circles.

Furthermore, the authors argued that (at the time) we did not have the computational resources required to construct large, deep neural networks (in hindsight, they were absolutely correct). This single paper alone *almost killed* neural network research.

Luckily, the backpropagation algorithm and the research by Werbos (1974) [15], Rumelhart (1986) [16], and LeCun (1998) [17] were able to resuscitate neural networks from what could have been an early demise. Their research in the backpropagation algorithm enabled *multi-layer feedforward* neural networks to be trained (Figure 2.4).

Combined with nonlinear activation functions, researchers could now learn nonlinear functions and solve the XOR problem, opening the gates to an entirely new area of research in neural networks. Further research demonstrated that neural networks are *universal approximators* [18], capable of approximating any continuous function (but placing no guarantee on whether or not the network can actually *learn* the parameters required to represent a function).

The backpropagation algorithm is the cornerstone of modern day neural networks allowing us to efficiently train neural networks and "teach" them to learn from their mistakes. But even so, at this time, due to (1) slow computers (compared to modern day machines) and (2) lack of large, labeled training sets, researchers were unable to (reliably) train neural networks that had more than two hidden layers – it was simply computationally infeasible.

Today, the latest incarnation of neural networks as we know it is called **deep learning**. What sets deep learning apart from its previous incarnations is that we have faster, specialized hardware with more available training data. We can now train networks with *many more hidden layers* that are capable of hierarchical learning where simple concepts are learned in the lower layers and more abstract patterns in the higher layers of the network.

Perhaps the quintessential example of applied deep learning to feature learning is the *Convo-*

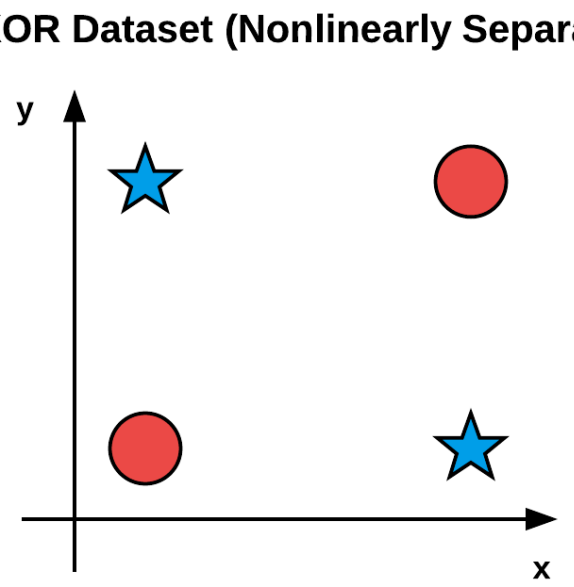


Figure 2.3: The XOR (E(X)clusive Or) dataset is an example of a nonlinear separable problem that the Perceptron *cannot* solve. Take a second to convince yourself that it is impossible to draw a single line that separates the blue stars from the red circles.

lutional Neural Network (LeCun 1988) [19] applied to handwritten character recognition which *automatically* learns discriminating patterns (called "filters") from images by sequentially stacking layers on top of each other. Filters in lower levels of the network represent edges and corners, while higher level layers use the edges and corners to learn more abstract concepts useful for discriminating between image classes.

In many applications, CNNs are now considered the most powerful image classifier and are currently responsible for pushing the state-of-the-art forward in computer vision subfields that leverage machine learning. For a more thorough review of the history of neural networks and deep learning, please refer to Goodfellow et al. [10] as well as this excellent blog post by Jason Brownlee at Machine Learning Mastery [20].

2.2 Hierarchical Feature Learning

Machine learning algorithms (generally) fall into three camps – *supervised*, *unsupervised*, and *semi-supervised* learning. We'll discuss supervised and unsupervised learning in this chapter while saving semi-supervised learning for a future discussion.

In the supervised case, a machine learning algorithm is given both a set of *inputs* and *target outputs*. The algorithm then tries to learn patterns that can be used to automatically map input data points to their correct target output. Supervised learning is similar to having a teacher watching you take a test. Given your previous knowledge, you do your best to mark the correct answer on your exam; however, if you are incorrect, your teacher guides you toward a better, more educated guess the next time.

In an unsupervised case, machine learning algorithms try to automatically discover discriminating features *without* any hints as to what the inputs are. In this scenario, our student tries to group similar questions and answers together, even though the student does not know what the correct answer is *and* the teacher is not there to provide them with the true answer. Unsupervised

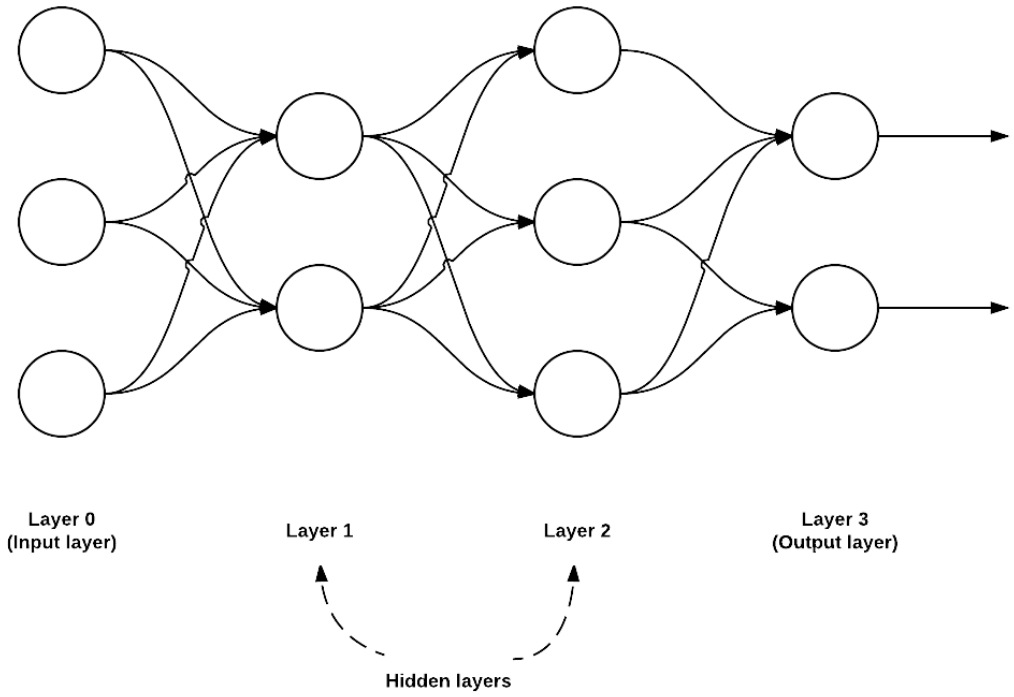


Figure 2.4: A multi-layer, feedforward network architecture with an input layer (3 nodes), two hidden layers (2 nodes in the first layer and 3 nodes in the second layer), and an output layer (2 nodes).

learning is clearly a more challenging problem than supervised learning – by knowing the answers (i.e., target outputs), we can more easily define discriminate patterns that can map input data to the correct target classification.

In the context of machine learning applied to image classification, the goal of a machine learning algorithm is to take these sets of images and identify patterns that can be used to discriminate various image classes/objects from one another.

In the past, we used *hand-engineered features* to quantify the contents of an image – we *rarely* used raw pixel intensities as inputs to our machine learning models, as is now common with deep learning. For each image in our dataset, we performed *feature extraction*, or the process of taking an input image, quantifying it according to some algorithm (called a *feature extractor* or *image descriptor*), and returning a vector (i.e., a list of numbers) that aimed to quantify the contents of an image. Figure 2.5 below depicts the process of quantifying an image containing prescription pill medication via a series of blackbox color, texture, and shape image descriptors.

Our hand-engineered features attempted to encode texture (Local Binary Patterns [21], Haralick texture [22]), shape (Hu Moments [23], Zernike Moments [24]), and color (color moments, color histograms, color correlograms [25]).

Other methods such as keypoint detectors (FAST [26], Harris [27], DoG [28], to name a few) and local invariant descriptors (SIFT [28], SURF [29], BRIEF [30], ORB [31], etc.) describe *salient* (i.e., the most "interesting") regions of an image.

Other methods such as Histogram of Oriented Gradients (HOG) [32] proved to be very good at detecting objects in images when the viewpoint angle of our image did not vary dramatically from what our classifier was trained on. An example of using the HOG + Linear SVM detector method

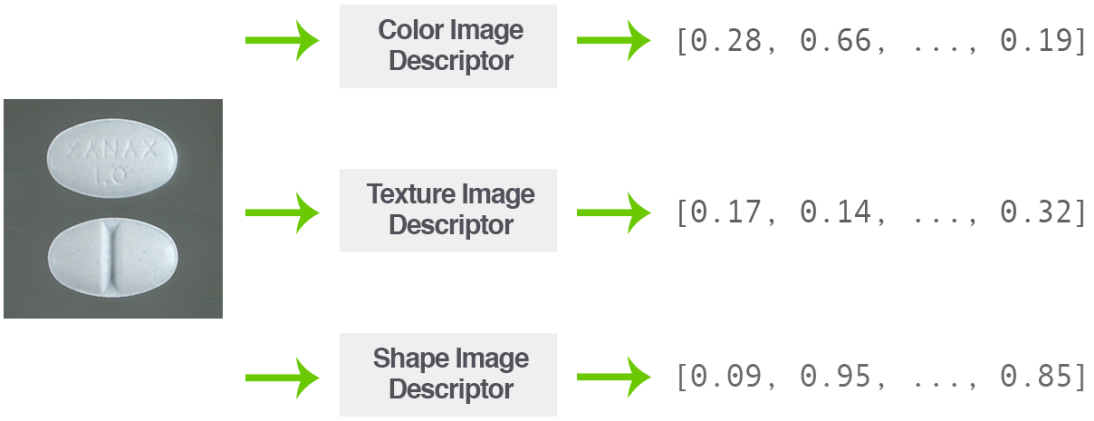


Figure 2.5: Quantifying the contents of an image containing a prescription pill medication via a series of blackbox color, texture, and shape image descriptors.

can be seen in Figure 2.6 where we detect the presence of stop signs in images.

For a while, research in object detection in images was guided by HOG and its variants, including computationally expensive methods such as the Deformable Parts Model [34] and Exemplar SVMs [35].

For a more in-depth study of image descriptors, feature extraction, and the process it plays in computer vision, be sure to refer to the PyImageSearch Gurus course [33].

In each of these situations, an algorithm was *hand-defined* to quantify and encode a particular aspect of an image (i.e., shape, texture, color, etc.). Given an input image of pixels, we would apply our hand-defined algorithm to the pixels, and in return receive a feature vector quantifying the image contents – the image pixels themselves did not serve a purpose other than being inputs to our feature extraction process. The feature vectors that resulted from feature extraction were what we were truly interested in as they served as inputs to our machine learning models.

Deep learning, and specifically Convolutional Neural Networks, take a different approach. Instead of hand-defining a set of rules and algorithms to extract features from an image, **these features are instead automatically learned from the training process**.

Again, let's return to the goal of machine learning: *computers should be able to learn from experience (i.e., examples) of the problem they are trying to solve*.

Using deep learning, we try to understand the problem in terms of a hierarchy of concepts. Each concept builds on top of the others. Concepts in the lower level layers of the network encode some basic representation of the problem, whereas higher level layers *use these basic layers* to form more abstract concepts. This hierarchical learning allows us to *completely remove* the hand-designed feature extraction process and treat CNNs as end-to-end learners.

Given an image, we supply the pixel intensity values as **inputs** to the CNN. A series of **hidden layers** are used to extract features from our input image. These hidden layers build upon each other in a hierarchal fashion. At first, only edge-like regions are detected in the lower level layers of the network. These edge regions are used to define corners (where edges intersect) and contours (outlines of objects). Combining corners and contours can lead to abstract "object parts" in the next layer.

Again, keep in mind that the types of concepts these filters are learning to detect are *automatically learned* – there is no intervention by us in the learning process. Finally, **output** layer is

Figure 2.6: The HOG + Linear SVM object detection framework applied to detecting the location of stop signs in images, as covered inside the PyImageSearch Gurus course [33].

used to classify the image and obtain the output class label – the output layer is either *directly* or *indirectly* influenced by every other node in the network.

We can view this process as hierarchical learning: each layer in the network uses the output of previous layers as "building blocks" to construct increasingly more abstract concepts. These layers are learned *automatically* – there is *no hand-crafted feature engineering* taking place in our network. Figure 2.7 compares classic image classification algorithms using hand-crafted features to representation learning via deep learning and Convolutional Neural Networks.

One of the primary benefits of deep learning and Convolutional Neural Networks is that it allows us to skip the feature extraction step and instead focus on process of training our network to learn these filters. However, as we'll find out later in this book, training a network to obtain reasonable accuracy on a given image dataset isn't always an easy task.

2.3 How "Deep" Is Deep?

To quote Jeff Dean from his 2016 talk, *Deep Learning for Building Intelligent Computer Systems* [36]:

> *"When you hear the term deep learning, just think of a large, deep neural net. Deep refers to the number of layers typically and so this kind of the popular term that's been adopted in the press."*

This is an excellent quote as it allows us to conceptualize deep learning as large neural networks where layers build on top of each other, gradually increasing in depth. **The problem is we still don't have a concrete answer to the question, "How many layers does a neural network need to be considered *deep?*"**

The short answer is there is ***no consensus*** amongst experts on the depth of a network to be considered deep [10].

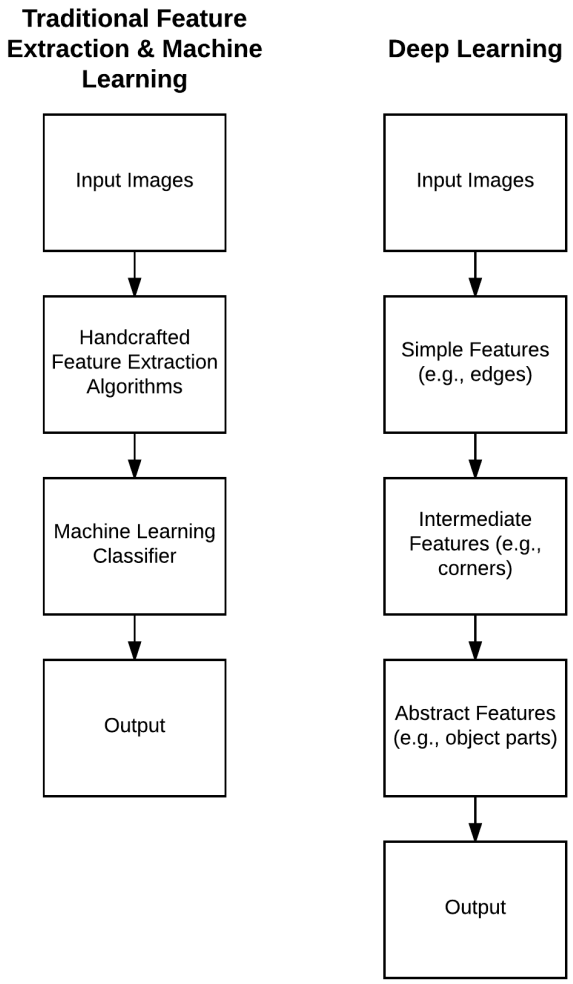


Figure 2.7: **Left:** Traditional process of taking an input set of images, applying hand-designed feature extraction algorithms, followed by training a machine learning classifier on the features. **Right:** Deep learning approach of stacking layers on top of each other that *automatically* learn more complex, abstract, and discriminating features.

And now we need to look at the question of network type. By definition, a Convolutional Neural Network (CNN) is a type of deep learning algorithm. But suppose we had a CNN with only one convolutional layer – is a network that is shallow, but yet still belongs to a family of algorithms inside the deep learning camp considered to be "deep"?

My personal opinion is that any network with greater than two hidden layers can be considered "deep". My reasoning is based on previous research in ANNs that were heavily handicapped by:

1. Our lack of large, labeled datasets available for training
2. Our computers being too slow to train large neural networks
3. Inadequate activation functions

Because of these problems, we could not easily train networks with more than two hidden layers during the 1980s and 1990s (and prior, of course). In fact, Geoff Hinton supports this sentiment in his 2016 talk, *Deep Learning* [37], where he discussed why the previous incarnations of deep learning (ANNs) did not take off during the 1990s phase:

1. Our labeled datasets were thousands of times too small.

2. Our computers were millions of times too slow.
3. We initialized the network weights in a stupid way.
4. We used the wrong type of nonlinearity activation function.

All of these reasons point to the fact that training networks with a depth larger than two hidden layers were a futile, if not a computational, impossibility.

In the current incarnation we can see that the tides have changed. We now have:

1. Faster computers
2. Highly optimized hardware (i.e., GPUs)
3. Large, labeled datasets in the order of millions of images
4. A better understanding of weight initialization functions and what does/does not work
5. Superior activation functions and an understanding regarding why previous nonlinearity functions stagnated research

Paraphrasing Andrew Ng from his 2013 talk, *Deep Learning, Self-Taught Learning and Unsupervised Feature Learning* [38], we are now able to construct deeper neural networks and train them with more data.

As the *depth* of the network increases, so does the *classification accuracy*. This behavior is different from traditional machine learning algorithms (i.e., logistic regression, SVMs, decision trees, etc.) where we reach a plateau in performance even as available training data increases. A plot inspired by Andrew Ng's 2015 talk, *What data scientists should know about deep learning*, [39] can be seen in Figure 2.8, providing an example of this behavior.

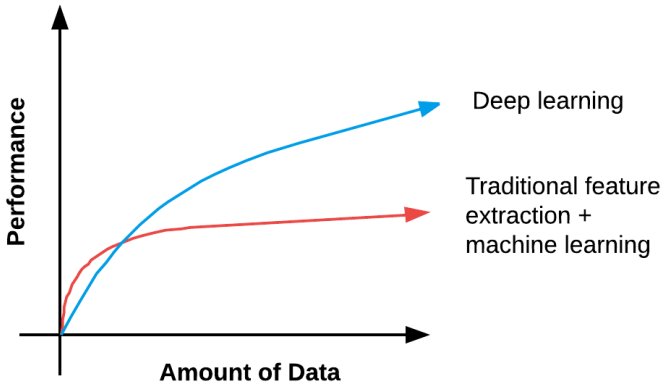


Figure 2.8: As the amount of data available to deep learning algorithms increases, accuracy does as well, substantially outperforming traditional feature extraction + machine learning approaches.

As the amount of training data increases, our neural network algorithms obtain higher classification accuracy, whereas previous methods plateau at a certain point. Because of the relationship between higher accuracy and more data, we tend to associate *deep learning* with *large datasets* as well.

When working on your own deep learning applications, I suggest using the following rule of thumb to determine if your given neural network is deep:

1. Are you using a *specialized* network architecture such as Convolutional Neural Networks, Recurrent Neural Networks, or Long Short-Term Memory (LSTM) networks? If so, **yes, you are performing deep learning**.
2. Does your network have a depth > 2? If yes, **you are doing deep learning**.
3. Does your network have a depth > 10? If so, **you are performing *very* deep learning** [40].

All that said, try not to get caught up in the buzzwords surrounding deep learning and what is/is not deep learning. At the very core, deep learning has gone through a number of different

incarnations over the past 60 years based on various schools of thought – **but *each* of these schools of thought centralize around artificial neural networks inspired by the structure and function of the brain**. Regardless of network depth, width, or specialized network architecture, you're *still* performing machine learning using artificial neural networks.

2.4 Summary

This chapter addressed the complicated question of *"What is deep learning?"*.

As we found out, deep learning has been around since the 1940s, going by different names and incarnations based on various schools of thought and popular research trends at a given time. At the very core, deep learning belongs to the family of Artificial Neural Networks (ANNs), a set of algorithms that learn patterns inspired by the structure and function of the brain.

There is no consensus amongst experts on exactly what makes a neural network "deep"; however, we know that:

1. Deep learning algorithms learn in a hierarchical fashion and therefore stack multiple layers on top of each other to learn increasingly more abstract concepts.
2. A network should have > 2 layers to be considered "deep" (this is my anecdotal opinion based on decades of neural network research).
3. A network with > 10 layers is considered *very deep* (although this number will change as architectures such as ResNet have been successfully trained with over 100 layers).

If you feel a bit confused or even overwhelmed after reading this chapter, don't worry – the purpose here was simply to provide an extremely high-level overview of deep learning and what exactly "deep" means.

This chapter also introduced a number of concepts and terms you may be unfamiliar with, including pixels, edges, and corners – our next chapter will address these types of image basics and give you a concrete foundation to stand on. We'll then start to move into the fundamentals of neural networks, allowing us to graduate to deep learning and Convolutional Neural Networks later in this book. While this chapter was admittedly high-level, the rest of the chapters of this book will be extremely hands-on, allowing you to master deep learning for computer vision concepts.

3. Image Fundamentals

Before we can start building our own image classifiers, we first need to understand what an image is. We'll start with the buildings blocks of an image – the pixel.

We'll discuss exactly what a pixel is, how they are used to form an image, and how to access pixels that are represented as NumPy arrays (as nearly all image processing libraries in Python do, including OpenCV and scikit-image).

The chapter will conclude with a discussion on the aspect ratio of an image and the relation it has when preparing our image dataset for training a neural network.

3.1 Pixels: The Building Blocks of Images

Pixels are the raw building blocks of an image. Every image consists of a set of pixels. There is no finer granularity than the pixel.

Normally, a pixel is considered the "color" or the "intensity" of light that appears in a given place in our image. If we think of an image as a grid, each square contains a single pixel. For example, take a look at Figure 3.1.

The image in Figure 3.1 above has a resolution of $1,000 \times 750$, meaning that it is $1,000$ pixels wide and 750 pixels tall. We can conceptualize an image as a (multidimensional) matrix. In this case, our matrix has $1,000$ columns (the width) with 750 rows (the height). Overall, there are $1,000 \times 750 = 750,000$ total pixels in our image.

Most pixels are represented in two ways:

1. Grayscale/single channel
2. Color

In a grayscale image, each pixel is a scalar value between 0 and 255, where zero corresponds to "black" and 255 being "white". Values between 0 and 255 are varying shades of gray, where values closer to 0 are darker and values closer to 255 are lighter. The grayscale gradient image in Figure 3.2 demonstrates *darker pixels* on the left-hand side and progressively *lighter pixels* on the right-hand side.

Color pixels; however, are normally represented in the RGB color space (other color spaces do exist, but are outside the scope of this book and not relevant for deep learning).

Figure 3.1: This image is 1,000 pixels wide and 750 pixels tall, for a total of $1,000 \times 750 = 750,000$ total pixels.

Figure 3.2: Image gradient demonstrating pixel values going from black (0) to white (255).

If you are interested in learning more about color spaces (and the fundamentals of computer vision and image processing), please see *Practical Python and OpenCV* along with the PyImageSearch Gurus course.

Pixels in the RGB color space are no longer a scalar value like in a grayscale/single channel image – instead, the pixels are represented by a list of *three values*: one value for the Red component, one for Green, and another for Blue. To define a color in the RGB color model, all we need to do is define the amount of Red, Green, and Blue contained in a single pixel.

Each Red, Green, and Blue channel can have values defined in the range $[0, 255]$ for a total of 256 "shades", where 0 indicates no representation and 255 demonstrates full representation. Given that the pixel value only needs to be in the range $[0, 255]$, we normally use 8-bit unsigned integers to represent the intensity.

As we'll see once we build our first neural network, we'll often preprocess our image by performing mean subtraction or scaling, which will require us to convert the image to a floating point data type. Keep this point in mind as the data types used by libraries loading images from disk (such as OpenCV) will often need to be converted before we apply learning algorithms to the images directly.

Given our three Red, Green, and Blue values, we can combine them into an RGB tuple in the form `(red, green, blue)`. This tuple represents a given color in the RGB color space. The RGB color space is an example of an *additive* color space: the more of each color is added, the brighter the pixel becomes and closer to white. We can visualize the RGB color space in Figure 3.3 (*left*). As you can see, adding red and green leads to yellow. Adding red and blue yields pink. And

adding all three red, green, and blue together, we create white.

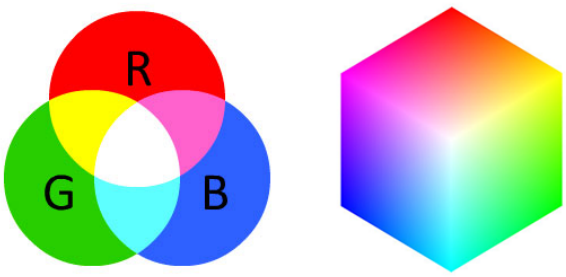


Figure 3.3: **Left:** The RGB color space is *additive*. The more red, green and blue you mix together, the closer you get to *white*. **Right:** The RGB cube.

To make this example more concrete, let's again consider the color "white" – we would fill each of the red, green, and blue buckets up completely, like this: `(255, 255, 255)`. Then, to create the color black, we would empty each of the buckets out `(0, 0, 0)`, as black is the absence of color. To create a pure red color, we would fill up the red bucket (and only the red bucket) completely: `(255, 0, 0)`.

The RGB color space is also commonly visualized as a cube (Figure 3.3, *right*) Since an RGB color is defined as a 3-valued tuple, which each value in the range $[0, 255]$ we can thus think of the cube containing $256 \times 256 \times 256 = 16,777,216$ possible colors, depending on how much Red, Green, and Blue are placed into each bucket.

As an example, let's consider how "much" red, green and blue we would need to create a single color (Figure 3.4, *top*). Here we set `R=252, G=198, B=188` to create a color tone similar to the skin of a caucasian (perhaps useful when building an application to detect the amount of skin/flesh in an image). As we can see, the Red component is heavily represented with the bucket almost filled. Green and Blue are represented almost equally. Combining these colors in an additive manner, we obtain a color tone similar to caucasian skin.

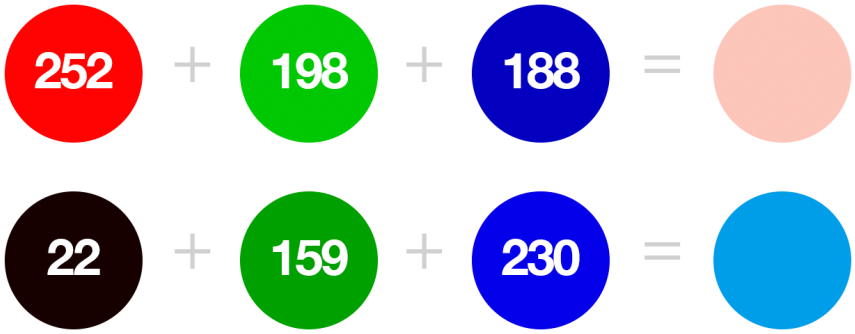


Figure 3.4: **Top:** An example of adding various Red, Green, and Blue color components together to create a "caucasian flesh tone", perhaps useful in a skin detection program. **Bottom:** Creating the specific shade of blue in the "PyImageSearch" logo by mixing various amounts of Red, Green, and Blue.

Let's try another example, this time setting `R=22, G=159, B=230` to obtain the shade of blue used in the PyImageSearch logo (Figure 3.4, *bottom*). Here Red is *heavily* under-represented with a

value of 22 – the bucket is so *empty* that the Red value actually appears to be "black". The Green bucket is a little over 50% full while the Blue bucket is over 90% filled, clearly the dominant color in the representation.

The primary drawbacks of the RGB color space include:

- Its additive nature makes it a bit unintuitive for humans to easily define shades of color without using a "color picker" tool.
- It doesn't mimic how humans perceive color.

Despite these drawbacks, nearly all images you'll work with will be represented (at least initially) in the RGB color space. For a full review of color models and color spaces, please refer to the PyImageSearch Gurus course [33].

3.1.1 Forming an Image From Channels

As we now know, an RGB image is represented by three values, one for each of the Red, Green, and Blue components, respectively. We can conceptualize an RGB image as consisting of *three independent matrices* of width `W` and height `H`, one for each of the RGB components, as shown in Figure 3.5. We can combine these three matrices to obtain a multi-dimensional array with shape $W \times H \times D$ where `D` is the **depth** or **number of channels** (for the RGB color space, `D=3`):

Figure 3.5: Representing an image in the RGB color space where each channel is an independent matrix, that when combined, forms the final image.

Keep in mind that the **depth** of an image is *very different* than the **depth** of a neural network – this point will become clear when we start training our own Convolutional Neural Networks. However, for the time being simply understand that the vast majority of the images you'll be working with are:

- Represented in the RGB color space by three channels, each channel in the range $[0, 255]$. A given pixel in an RGB image is a list of three integers: one value for Red, second for Green, and a final value for Blue.
- Programmatically defined as a 3D NumPy multidimensional arrays with a width, height, and depth.

3.2 The Image Coordinate System

As mentioned in Figure 3.1 earlier in this chapter, an image is represented as a grid of pixels. To make this point more clear, imagine our grid as a piece of graph paper. Using this graph paper, the origin point $(0, 0)$ corresponds to the **upper-left** corner of the image. As we move down and to the right, both the x and y values increase.

Figure 3.6 provides a visual representation of this "graph paper" representation. Here we have the letter "I" on a piece of our graph paper. We see that this is an 8×8 grid with a total of 64 pixels.

It's important to note that we are counting from *zero* rather than *one*. The Python language is *zero indexed*, meaning that we always start counting from zero. Keep this in mind as you'll avoid a lot of confusion later on (especially if you're coming from a MATLAB environment).

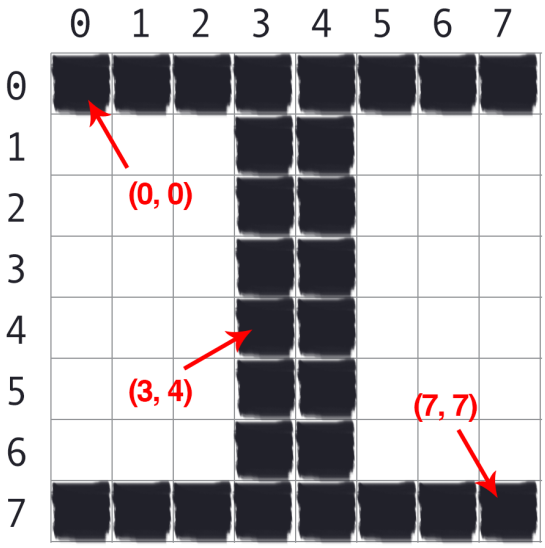


Figure 3.6: The letter "I" placed on a piece of graph paper. Pixels are accessed by their (x,y)-coordinates, where we go x columns to the right and y rows down, keeping in mind that Python is zero-indexed.

As an example of zero-indexing, consider the pixel 4 columns to the right and 5 rows down is indexed by the point $(3,4)$, again keeping in mind that we are counting from *zero* rather than *one*.

3.2.1 Images as NumPy Arrays

Figure 3.7: Loading an image named `example.png` from disk and displaying it to our screen with OpenCV.

Image processing libraries such as OpenCV and scikit-image represent RGB images as multi-dimensional NumPy arrays with shape `(height, width, depth)`. Readers who are using image processing libraries for the first time are often confused by this representation – why does the *height* come before the *width* when we normally think of an image in terms of width *first* then height?

The answer is due to matrix notation.

When defining the dimensions of matrix, we always write it as `rows x columns`. The number of `rows` in an image is its height whereas the number of `columns` is the image's width. The depth will still remain the depth.

Therefore, while it may be slightly confusing to see the `.shape` of a NumPy array represented as `(height, width, depth)`, this representation actually makes intuitive sense when considering how a matrix is constructed and annotated.

For example, let's take a look at the OpenCV library and the `cv2.imread` function used to load an image from disk and display its dimensions:

```
1 import cv2
2 image = cv2.imread("example.png")
3 print(image.shape)
4 cv2.imshow("Image", image)
5 cv2.waitKey(0)
```

Here we load an image named `example.png` from disk and display it to our screen, as the screenshot from Figure 3.7 demonstrates. My terminal output follows:

```
$ python load_display.py
(248, 300, 3)
```

This image has a width of 300 pixels (the number of columns), a height of 248 pixels (the number of rows), and a depth of 3 (the number of channels). To access an individual pixel value from our `image` we use simple NumPy array indexing:

```
1 (b, g, r) = image[20, 100] # accesses pixel at x=100, y=20
2 (b, g, r) = image[75, 25] # accesses pixel at x=25, y=75
3 (b, g, r) = image[90, 85] # accesses pixel at x=85, y=90
```

Again, notice how the `y` value is passed in *before* the `x` value – this syntax may feel uncomfortable at first, but it is consistent with how we access values in a matrix: first we specify the row number then the column number. From there, we are given a tuple representing the Red, Green, and Blue components of the image.

3.2.2 RGB and BGR Ordering

It's important to note that OpenCV stores RGB channels in *reverse order*. While we normally think in terms of Red, Green, and Blue, OpenCV actually stores the pixel values in Blue, Green, Red order.

Why does OpenCV do this? The answer is simply historical reasons. Early developers of the OpenCV library chose the BGR color format because the BGR ordering was popular among camera manufacturers and other software developers at the time [41].

Simply put – this BGR ordering was made for historical reasons and a choice that we now have to live with. It's a small caveat, but an important one to keep in mind when working with OpenCV.

3.3 Scaling and Aspect Ratios

Scaling, or simply *resizing*, is the process of increasing or decreasing the size of an image in terms of width and height. When resizing an image, it's important to keep in mind the *aspect ratio*, which

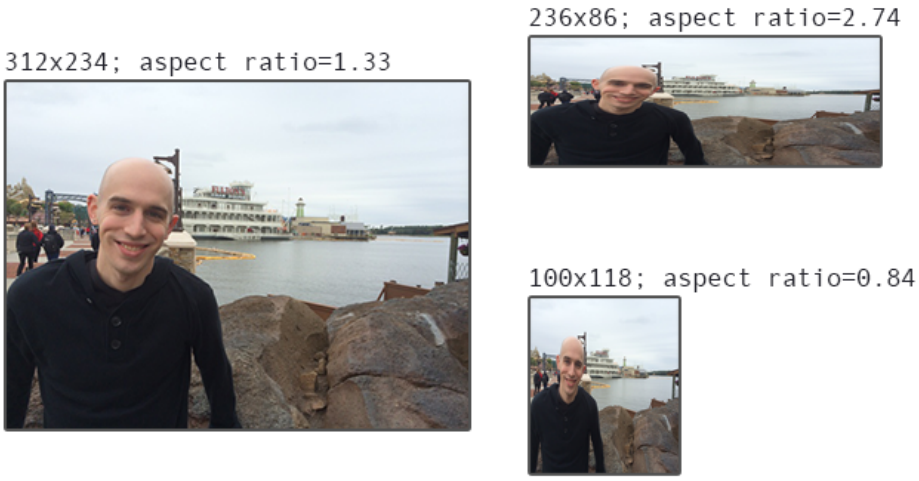


Figure 3.8: **Left:** Original image. **Top and Bottom:** Resulting distorted images after resizing without preserving the aspect ratio (i.e., the ratio of the width to the height of the image).

is the ratio of the width to the height of the image. Ignoring the aspect ratio can lead to images that look compressed and distorted, as in Figure 3.8.

On the *left*, we have the original image. And on the *top and bottom*, we have two images that have been distorted by not preserving the aspect ratio. The end result is that these images are distorted, crunched, and squished. To prevent this behavior, we simply scale the width and height of an image by equal amounts when resizing an image.

From a strictly *aesthetic* point of view, you almost always want to ensure the aspect ratio of the image is maintained when resizing an image – **but this guideline isn't always the case for deep learning**. Most neural networks and Convolutional Neural Networks applied to the task of image classification assume a *fixed size input*, meaning that the dimensions of *all images* you pass through the network *must be the same*. Common choices for width and height image sizes inputted to Convolutional Neural Networks include 32×32, 64×64, 224×224, 227×227, 256×256, and 299×299.

Let's assume we are designing a network that will need to classify 224×224 images; however, our dataset consists of images that are 312×234, 800×600, and 770×300, among other image sizes – how are we supposed to preprocess these images? Do we simply ignore the aspect ratio and deal with the distortion (Figure 3.9, *bottom left*)? Or do we devise another scheme to resize the image, such as resizing the image along its shortest dimension and then taking the center crop (Figure 3.9, *bottom right*)?

As we can see in in the *bottom left*, the aspect ratio of our image has been ignored, resulting in an image that looks distorted and "crunched". Then, in the *bottom right*, we see that the aspect ratio of the image has been maintained, but at the expense of cropping out part of the image. This could be especially detrimental to our image classification system if we accidentally crop part or all of the object we wish to identify.

Which method is best? **In short, *it depends.*** For some datasets you can simply ignore the aspect ratio and squish, distort, and compress your images prior to feeding them through your network. On other datasets, it's advantageous to preprocess them further by resizing along the shortest dimension and then cropping the center.

We'll be reviewing both of these methods (and how to implement them) in more detail later in this book, but it's important to introduce this topic now as we study the fundamentals of images

Figure 3.9: **Top:** Our original input image. **Bottom left:** Resizing an image to 224×224 pixels by ignoring the aspect ratio. **Bottom right:** Resizing an image 224×224 pixels by first resizing along the shortest dimension and then taking the center crop.

and how they are represented.

3.4 Summary

This chapter reviewed the fundamental building blocks of an image – the pixel. We learned that grayscale/single channel images are represented by a single scalar, the intensity/brightness of the pixel. The most common color space is the RGB color space where each pixel in an image is represented by a 3-tuple: one for each of the Red, Green, and Blue components, respectively.

We also learned that computer vision and image processing libraries in the Python programming language leverage the powerful NumPy numerical processing library and thus represent images as multi-dimensional NumPy arrays. These arrays have the shape `(height, width, depth)`.

The height is specified first because the height is the number of rows in the matrix. The width comes next, as it is the number of columns in the matrix. Finally, the depth controls the number of channels in the image. In the RGB color space, the depth is fixed at `depth=3`.

Finally, we wrapped up this chapter by reviewing the *aspect ratio* of an image and the role it will play when we resize images as inputs to our neural networks and Convolutional Neural Networks. For a more detailed review of color spaces, the image coordinate system, resizing/aspect ratios, and other basics of the OpenCV library, please refer to *Practical Python and OpenCV* [8] and the PyImageSearch Gurus course [33].

4. Image Classification Basics

"A picture is worth a thousand words" – English idiom

We've heard this adage countless times in our lives. It simply means that a complex idea can be conveyed in a single image. Whether examining the line chart of our stock portfolio investments, looking at the spread of an upcoming football game, or simply taking in the art and brush strokes of a painting master, we are constantly ingesting visual content, interpreting the meaning, and storing the knowledge for later use.

However, for computers, interpreting the contents of an image is less trivial – all our computer sees is a big matrix of numbers. It has *no idea* regarding the thoughts, knowledge, or meaning the image is trying to convey.

In order to understand the contents of an image, we must apply ***image classification***, which is the task of using computer vision and machine learning algorithms to extract meaning from an image. This action could be as simple as assigning a label to what the image contains, or as advanced as interpreting the contents of an image and returning a human-readable sentence.

Image classification is a very large field of study, encompassing a wide variety of techniques – and with the popularity of deep learning, it is continuing to grow.

Now is the time to ride the deep learning and image classification wave – those who successfully do so will be handsomely rewarded.

Image classification and image understanding are currently (and will continue to be) the most popular sub-field of computer vision for the next ten years. In the future, we'll see companies like Google, Microsoft, Baidu, and others quickly acquire successful image understanding startup companies. We'll see more and more consumer applications on our smartphones that can understand and interpret the contents of an image. Even wars will likely be fought using unmanned aircrafts that are *automatically* guided using computer vision algorithms.

Inside this chapter, I'll provide a high-level overview of what image classification is, along with the many challenges an image classification algorithm has to overcome. We'll also review the three different types of learning associated with image classification and machine learning.

Finally, we'll wrap up this chapter by discussing the four steps of training a deep learning network for image classification and how this four step pipeline compares to the *traditional,*

hand-engineered feature extraction pipeline.

4.1 What Is Image Classification?

Image classification, at its very core, is the task of *assigning a label to an image* from a *predefined set of categories*.

Practically, this means that our task is to analyze an input image and return a label that categorizes the image. The label is always from a predefined set of possible categories.

For example, let's assume that our set of possible categories includes:

```
categories = {cat, dog, panda}
```

Then we present the following image (Figure 4.1) to our classification system:

Figure 4.1: The goal of an image classification system is to take an input image and assign a label based on a pre-defined set of categories.

Our goal here is to take this input image and assign a label to it from our **categories** set – in this case, **dog**.

Our classification system could also assign multiple labels to the image via probabilities, such as **dog: 95%**; **cat: 4%**; **panda: 1%**.

More formally, given our input image of $W \times H$ pixels with three channels, Red, Green, and Blue, respectively, our goal is to take the $W \times H \times 3 = N$ pixel image and figure out how to correctly classify the contents of the image.

4.1.1 A Note on Terminology

When performing machine learning and deep learning, we have a ***dataset*** we are trying to extract knowledge from. Each example/item in the dataset (whether it be image data, text data, audio data, etc.) is a ***data point***. A dataset is therefore a collection of data points (Figure 4.2).

Our goal is to apply a machine learning and deep learning algorithms to discover underlying patterns in the dataset, enabling us to correctly classify data points that our algorithm has not encountered yet. Take the time now to familiarize yourself with this terminology:

1. In the context of image classification, our *dataset* is a *collection of images*.
2. Each *image* is, therefore, a *data point*.

I'll be using the term *image* and *data point* interchangeably throughout the rest of this book, so keep this in mind now.

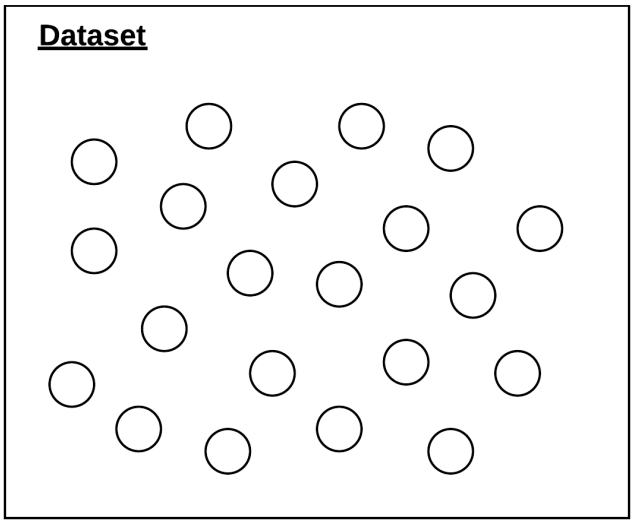


Figure 4.2: A dataset (outer rectangle) is a *collection* of data points (circles).

4.1.2 The Semantic Gap

Take a look at the two photos (*top*) in Figure 4.3. It should be fairly trivial for us to tell the difference between the two photos – there is clearly a *cat* on the left and a *dog* on the right. But all a computer sees is two big matrices of pixels (*bottom*).

Given that all a computer sees is a big matrix of pixels, we arrive at the problem of the *semantic gap*. The semantic gap is the difference between how a human *perceives* the contents of an image versus how an image can be *represented* in a way a computer can understand the process.

Again, a quick visual examination of the two photos above can reveal the difference between the two species of an animal. But in reality, the computer has *no idea* there are animals in the image to begin with. To make this point clear, take a look at Figure 4.4, containing a photo of a tranquil beach.

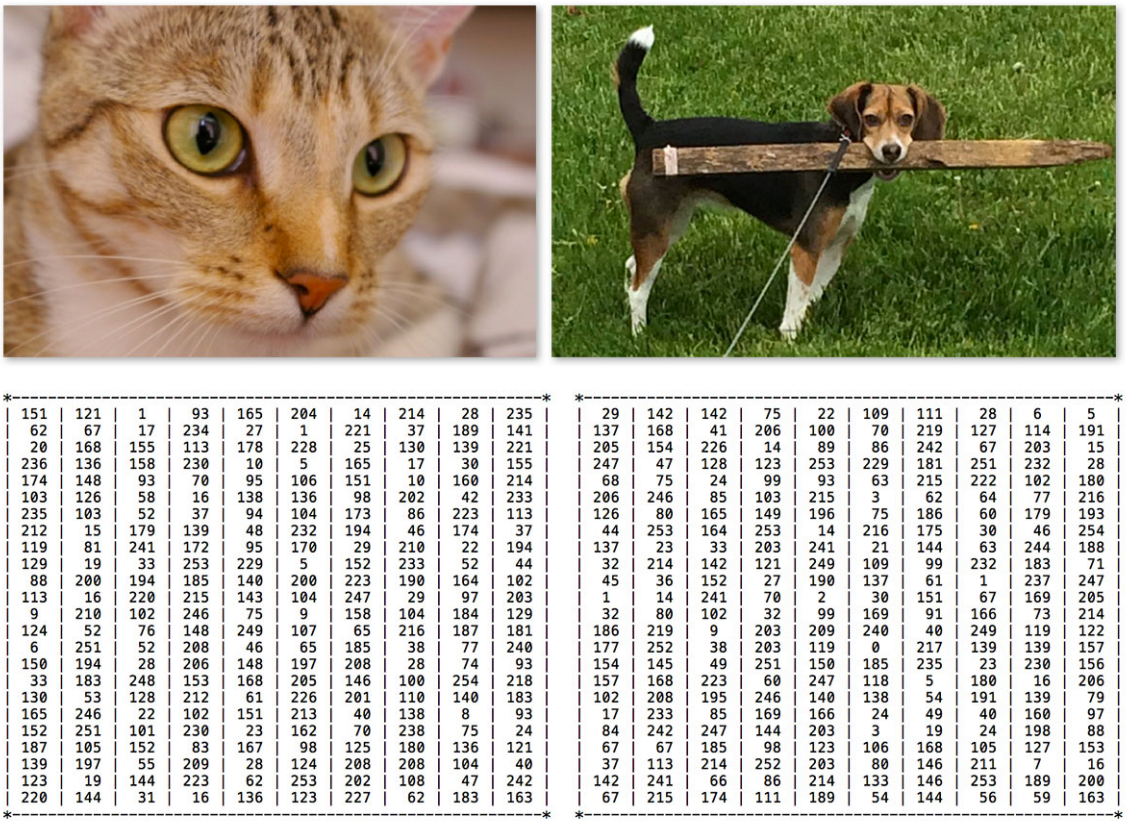
We might describe the image as follows:

- **Spatial:** The sky is at the top of the image and the sand/ocean are at the bottom.
- **Color:** The sky is dark blue, the ocean water is a lighter blue than the sky, while the sand is tan.
- **Texture:** The sky has a relatively uniform pattern, while the sand is very coarse.

How do we go about encoding all this information in a way that a computer can understand it? The answer is to apply *feature extraction* to quantify the contents of an image. Feature extraction is the process of taking an input image, applying an algorithm, and obtaining a feature vector (i.e., a list of numbers) that quantifies our image.

To accomplish this process, we may consider applying hand-engineered features such as HOG, LBPs, or other "traditional" approaches to image quantifying. Another method, and the one taken by this book, is to apply deep learning to automatically *learn a set of features* that can be used to quantify and ultimately *label* the contents of the image itself.

However, it's not that simple... because once we start examining images in the real world, we are faced with many, *many* challenges.



151	121	1	93	165	204	14	214	28	235
62	67	17	234	27	1	221	37	189	141
20	168	155	113	178	228	25	130	139	221
236	136	158	230	10	5	165	17	30	155
174	148	93	70	95	106	151	10	160	214
103	126	58	16	138	136	98	202	42	233
235	103	52	37	94	104	173	86	223	113
212	15	179	139	48	232	194	46	174	37
119	81	241	172	95	170	29	210	22	194
129	19	33	253	229	5	152	233	52	44
88	200	194	185	140	200	223	190	164	102
113	16	220	215	143	104	247	29	97	203
9	210	102	246	75	9	158	104	184	129
124	52	76	148	249	107	65	216	187	181
6	251	52	208	46	65	185	38	77	240
150	194	28	206	148	197	208	28	74	93
33	183	248	153	168	205	146	100	254	218
130	53	128	212	61	226	201	110	140	183
165	246	22	102	151	213	40	138	8	93
152	251	101	230	23	162	70	238	75	24
187	105	152	83	167	98	125	180	136	121
139	197	55	209	28	124	208	208	104	40
123	19	144	223	62	253	202	108	47	242
220	144	31	16	136	123	227	62	183	163

29	142	142	75	22	109	111	28	6	5
137	168	41	206	100	70	219	127	114	191
205	154	226	14	89	86	242	67	203	15
247	47	128	123	253	229	181	251	232	28
68	75	24	99	93	63	215	222	102	180
206	246	85	103	215	3	62	64	77	216
126	80	165	149	196	75	186	60	179	193
44	253	164	253	14	216	175	30	46	254
137	23	33	203	241	21	144	63	244	188
32	214	142	121	249	109	99	232	183	71
45	36	152	27	190	137	61	1	237	247
1	14	241	70	2	30	151	67	169	205
32	80	102	32	99	169	91	166	73	214
186	219	9	203	209	240	40	249	119	122
177	252	38	203	119	0	217	139	139	157
154	145	49	251	150	185	235	23	230	156
157	168	223	60	247	118	5	180	16	206
102	208	195	246	140	138	54	191	139	79
17	233	85	169	166	24	49	40	160	97
84	242	247	144	203	3	19	24	198	88
67	67	185	98	123	106	168	105	127	153
37	113	214	252	203	80	146	211	7	16
142	241	66	86	214	133	146	253	189	200
67	215	174	111	189	54	144	56	59	163

Figure 4.3: **Top:** Our brains can clearly see the difference between an image that contains a *cat* and an image that contains a *dog*. **Bottom:** However, all a computer "sees" is a big matrix of numbers. The difference between how we perceive an image and how the image is represented (a matrix of numbers) is called the *semantic gap*.

4.1.3 Challenges

If the semantic gap were not enough of a problem, we also have to handle **factors of variation** [10] in how an image or object appears. Figure 4.5 displays a visualization of a number of these factors of variation.

To start, we have ***viewpoint variation***, where an object can be oriented/rotated in multiple dimensions with respect to how the object is photographed and captured. No matter the angle in which we capture this Raspberry Pi, it's still a Raspberry Pi.

We also have to account for ***scale variation*** as well. Have you ever ordered a tall, grande, or venti cup of coffee from Starbucks? Technically they are all the same thing – a cup of coffee. But they are all different *sizes* of a cup of coffee. Furthermore, that same venti coffee will look dramatically different when it is photographed up close versus when it is captured from farther away. Our image classification methods must be tolerable to these types of scale variations.

One of the hardest variations to account for is ***deformation***. For those of you familiar with the television series *Gumby*, we can see the main character in the image above. As the name of the TV show suggests, this character is elastic, stretchable, and capable of contorting his body in many different poses. We can look at these images of *Gumby* as a type of *object deformation* – all images contain the *Gumby* character; however, they are all dramatically different from each other.

Our image classification should also be able to handle ***occlusions***, where large parts of the

Figure 4.4: When describing the contents of this image we may focus on words that the convey the *spatial layout*, *color*, and *texture* – the same is true for computer vision algorithms.

object we want to classify are hidden from view in the image (Figure 4.5). On the *left* we have to have a picture of a dog. And on the *right* we have a photo of the same dog, but notice how the dog is resting underneath the covers, *occluded* from our view. The dog is still clearly in both images – she's just more visible in one image than the other. Image classification algorithms should still be able to detect and label the presence of the dog in both images.

Just as challenging as the *deformations* and *occlusions* mentioned above, we also need to handle the changes in ***illumination***. Take a look at the coffee cup captured in standard and low lighting (Figure 4.5). The image on the *left* was photographed with standard overhead lighting while the image on the *right* was captured with very little lighting. We are still examining the same cup – but based on the lighting conditions, the cup looks dramatically different (nice how the vertical cardboard seam of the cup is clearly visible in the low lighting conditions, but not the standard lighting).

Continuing on, we must also account for ***background clutter***. Ever play a game of *Where's Waldo?* (Or *Where's Wally?* for our international readers.) If so, then you know the goal of the game is to find our favorite red-and-white, striped shirt friend. However, these puzzles are more than just an entertaining children's game – they are also the perfect representation of *background clutter*. These images are incredibly "noisy" and have a lot going on in them. We are only interested in *one* particular object in the image; however, due to all the "noise", it's not easy to pick out Waldo/Wally. If it's not easy for us to do, imagine how hard it is for a computer with no semantic understanding of the image!

Finally, we have ***intra-class variation***. The canonical example of intra-class variation in computer vision is displaying the diversification of chairs. From comfy chairs that we use to curl up and read a book, to chairs that line our kitchen table for family gatherings, to ultra-modern art deco chairs found in prestigious homes, a chair is still a chair – and our image classification algorithms must be able to categorize all these variations correctly.

Are you starting to feel a bit overwhelmed with the complexity of building an image classifier? Unfortunately, it only gets worse – it's not enough for our image classification system to be robust to these variations *independently*, but our system must also handle *multiple variations combined together!*

So how do we account for such an incredible number of variations in objects/images? In general, we try to frame the problem as best we can. We make assumptions regarding the contents of our images and to which variations we want to be tolerant. We also consider the scope of our

Figure 4.5: When developing an image classification system, we need to be cognizant of how an object can appear at varying viewpoints, lighting conditions, occlusions, scale, etc.

project – what is the end goal? And what are we trying to build?

Successful computer vision, image classification, and deep learning systems deployed to the real-world make *careful assumptions and considerations* before a single line of code is ever written.

If you take too broad of an approach, such as *"I want to classify and detect every single object in my kitchen"*, (where there could be hundreds of possible objects) then your classification system is unlikely to perform well unless you have years of experience building image classifiers – and even then, there is no guarantee to the success of the project.

But if you **frame your problem** and make it narrow in scope, such as *"I want to recognize* ***just*** *stoves and refrigerators"*, then your system is **much more likely** to be accurate and functioning, *especially* if this is your first time working with image classification and deep learning.

The key takeaway here is to ***always consider the scope of your image classifier***. While deep learning and Convolutional Neural Networks have demonstrated significant robustness and classification power under a variety of challenges, you *still* should keep the scope of your project as

tight and well-defined as possible.

Keep in mind that ImageNet [42], the *de facto* standard benchmark dataset for image classification algorithms, consists of 1,000 objects that we encounter in our everyday lives – and this dataset is *still* actively used by researchers trying to push the state-of-the-art for deep learning forward.

Deep learning is *not* magic. Instead, deep learning is like a scroll saw in your garage – powerful and useful when wielded correctly, but hazardous if used without proper consideration. Throughout the rest of this book, I will guide you on your deep learning journey and help point out when you should reach for these power tools and when you should instead refer to a simpler approach (or mention if a problem isn't reasonable for image classification to solve).

4.2 Types of Learning

There are three types of learning that you are likely to encounter in your machine learning and deep learning career: supervised learning, unsupervised learning, and semi-supervised learning. This book focuses mostly on supervised learning in the context of deep learning. Nonetheless, descriptions of all three types of learning are presented below.

4.2.1 Supervised Learning

Imagine this: you've just graduated from college with your Bachelors of Science in Computer Science. You're young. Broke. And looking for a job in the field – perhaps you even feel lost in your job search.

But before you know it, a Google recruiter finds you on LinkedIn and offers you a position working on their Gmail software. Are you going to take it? Most likely.

A few weeks later, you pull up to Google's spectacular campus in Mountain View, California, overwhelmed by the breathtaking landscape, the fleet of Teslas in the parking lot, and the almost never-ending rows of gourmet food in the cafeteria.

You finally sit down at your desk in a wide-open workspace among hundreds of other employees... *and then you find out your role in the company*. You've been hired to create a piece of software to *automatically classify* email as *spam* or *not-spam*.

How are going to accomplish this goal? Would a rule-based approach work? Could you write a series of `if/else` statements that look for certain words and then determine if an email is spam based on these rules? That might work... to a degree. But this approach would also be easily defeated and near impossible to maintain.

Instead, what you *really need* is machine learning. You need a *training set* consisting of the emails themselves along with their *labels*, in this case *spam* or *not-spam*. Given this data, you can analyze the text (i.e., the distributions of words) in the email and utilize the spam/not-spam labels to teach a machine learning classifier what words occur in a spam email and which do not – all without having to manually create a long and complicated series of `if/else` statements.

This example of creating a spam filter system is an example of **supervised learning**. Supervised learning is arguably the most well known and studied type of machine learning. Given our training data, a model (or "classier") is created through a training process where predictions are made on the input data and then corrected when the predictions are wrong. This training process continues until the model achieves some desired stopping criterion, such as a low error rate or a maximum number of training iterations.

Common supervised learning algorithms include Logistic Regression, Support Vector Machines (SVMs) [43, 44], Random Forests [45], and Artificial Neural Networks.

In the context of **image classification**, we assume our image dataset consists of the images themselves along with their corresponding *class label* that we can use to teach our machine learning

Label	R_μ	G_μ	B_μ	R_σ	G_σ	B_σ
Cat	57.61	41.36	123.44	158.33	149.86	93.33
Cat	120.23	121.59	181.43	145.58	69.13	116.91
Cat	124.15	193.35	65.77	23.63	193.74	162.70
Dog	100.28	163.82	104.81	19.62	117.07	21.11
Dog	177.43	22.31	149.49	197.41	18.99	187.78
Dog	149.73	87.17	187.97	50.27	87.15	36.65

Table 4.1: A table of data containing both the class labels (either *dog* or *cat*) and feature vectors for each data point (the mean and standard deviation of each Red, Green, and Blue color channel, respectively). This is an example of a ***supervised classification*** task.

classifier what each category "looks like". If our classifier makes an incorrect prediction, we can then apply methods to correct its mistake.

The differences between supervised, unsupervised, and semi-supervised learning can best be understood by looking at the example in Table 4.1. The first column of our table is the label associated with a particular image. The remaining six columns correspond to our feature vector for each data point – here, we have chosen to quantify our image contents by computing the mean and standard deviation for each RGB color channel, respectively.

Our supervised learning algorithm will make predictions on each of these feature vectors, and if it makes an incorrect prediction, we'll attempt to correct it by telling it what the correct label actually is. This process will then continue until the desired stopping criterion has been met, such as accuracy, number of iterations of the learning process, or simply an arbitrary amount of wall time.

To explain the differences between supervised, unsupervised, and semi-supervised learning, I have chosen to use a feature-based approach (i.e., the mean and standard deviation of the RGB color channels) to quantify the content of an image. When we start working with Convolutional Neural Networks, we'll actually **skip** the feature extraction step and use the raw pixel intensities themselves. Since images can be large $M \times N$ matrices (and therefore cannot fit nicely into this spreadsheet/table example), I have used the feature-extraction process to help visualize the differences between types of learning.

4.2.2 Unsupervised Learning

In contrast to supervised learning, **unsupervised learning** (sometimes called **self-taught learning**) has no labels associated with the input data and thus we cannot correct our model if it makes an incorrect prediction.

Going back to the spreadsheet example, converting a supervised learning problem to an unsupervised learning one is as simple as removing the "label" column (Table 4.2).

Unsupervised learning is sometimes considered the "holy grail" of machine learning and image classification. When we consider the number of images on Flickr or the number of videos on YouTube, we quickly realize there is a *vast amount* of unlabeled data available on the internet. If we could get our algorithm to learn patterns from *unlabeled data*, then we wouldn't have to spend large amounts of time (and money) arduously labeling images for supervised tasks.

Most unsupervised learning algorithms are most successful when we can learn the underlying structure of a dataset and then, in turn, apply our learned features to a *supervised* learning problem where there is too little labeled data to be of use.

Classic machine learning algorithms for unsupervised learning include Principle Component Analysis (PCA) and k-means clustering. Specific to neural networks, we see Autoencoders, Self-

R_μ	G_μ	B_μ	R_σ	G_σ	B_σ
57.61	41.36	123.44	158.33	149.86	93.33
120.23	121.59	181.43	145.58	69.13	116.91
124.15	193.35	65.77	23.63	193.74	162.70
100.28	163.82	104.81	19.62	117.07	21.11
177.43	22.31	149.49	197.41	18.99	187.78
149.73	87.17	187.97	50.27	87.15	36.65

Table 4.2: Unsupervised learning algorithms attempt to learn underlying patterns in a dataset *without* class labels. In this example we have removed the class label column, thus turning this task into an ***unsupervised learning*** problem.

Label	R_μ	G_μ	B_μ	R_σ	G_σ	B_σ
Cat	57.61	41.36	123.44	158.33	149.86	93.33
?	120.23	121.59	181.43	145.58	69.13	116.91
?	124.15	193.35	65.77	23.63	193.74	162.70
Dog	100.28	163.82	104.81	19.62	117.07	21.11
?	177.43	22.31	149.49	197.41	18.99	187.78
Dog	149.73	87.17	187.97	50.27	87.15	36.65

Table 4.3: When performing ***semi-supervised learning*** we only have the labels for a subset of the images/feature vectors and must try to label the other data points to utilize them as extra training data.

Organizing Maps (SOMs), and Adaptive Resonance Theory applied to unsupervised learning. Unsupervised learning is an extremely active area of research and one that has yet to be solved. We do not focus on unsupervised learning in this book.

4.2.3 Semi-supervised Learning

So, what happens if we only have *some* of the labels associated with our data and *no labels* for the other? Is there a way we can apply some hybrid of supervised and unsupervised learning and still be able to classify each of the data points? It turns out the answer is *yes* – we just need to apply semi-supervised learning.

Going back to our spreadsheet example, let's say we only have labels for a small fraction of our input data (Table 4.3). Our semi-supervised learning algorithm would take the known pieces of data, analyze them, and try to label each of the unlabeled data points for use as *additional* training data. This process can repeat for many iterations as the semi-supervised algorithm learns the "structure" of the data to make more accurate predictions and generate more reliable training data.

Semi-supervised learning is especially useful in computer vision where it is often time-consuming, tedious, and expensive (at least in terms of man-hours) to label each and every single image in our training set. In cases where we simply do not have the time or resources to label each individual image, we can label only a tiny fraction of our data and utilize semi-supervised learning to label and classify the rest of the images.

Semi-supervised learning algorithms often trade smaller labeled input datasets for some tolerable reduction in classification accuracy. Normally, the more accurately-labeled training a supervised learning algorithm has, the more accurate predictions it can make (this is *especially* true for deep learning algorithms).

As the amount of training data decreases, accuracy inevitably suffers. Semi-supervised learning

takes this relationship between accuracy and amount of data into account and attempts to keep classification accuracy within tolerable limits while dramatically reducing the amount of training data required to build a model – the end result is an accurate classifier (but normally not as accurate as a supervised classifier) with less effort and training data. Popular choices for semi-supervised learning include label spreading [46], label propagation [47], ladder networks [48], and co-learning/co-training [49].

Again, we'll primarily be focusing on supervised learning inside this book, as both unsupervised and semi-supervised learning in the context of deep learning for computer vision are still very active research topics without clear guidelines on which methods to use.

4.3 The Deep Learning Classification Pipeline

Based on our previous two sections on image classification and types of learning algorithms, you might be starting to feel a bit steamrolled with new terms, considerations, and what looks to be an insurmountable amount of variation in building an image classifier, but the truth is that building an image classifier is fairly straightforward, *once you understand the process*.

In this section we'll review an important shift in mindset you need to take on when working with machine learning. From there I'll review the four steps of building a deep learning-based image classifier as well as compare and contrast traditional feature-based machine learning versus end-to-end deep learning.

4.3.1 A Shift in Mindset

Before we get into anything complicated, let's start off with something that we're all (most likely) familiar with: *the Fibonacci sequence*.

The Fibonacci sequence is a series of numbers where the next number of the sequence is found by summing the two integers before it. For example, given the sequence `0, 1, 1`, the next number is found by adding `1 + 1 = 2`. Similarly, given `0, 1, 1, 2`, the next integer in the sequence is `1 + 2 = 3`.

Following that pattern, the first handful of numbers in the sequence are as follows:

`0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`

Of course, we can also define this pattern in an (extremely unoptimized) Python function using recursion:

```
>>> def fib(n):
...     if n == 0:
...             return 0
...     elif n == 1:
...             return 1
...     else:
...             return fib(n-1) + fib(n-2)
...
>>>
```

Using this code, we can compute the n-th number in the sequence by supplying a value of `n` to the `fib` function. For example, let's compute the 7th number in the Fibonacci sequence:

```
>>> fib(7)
13
```

And the 13th number: