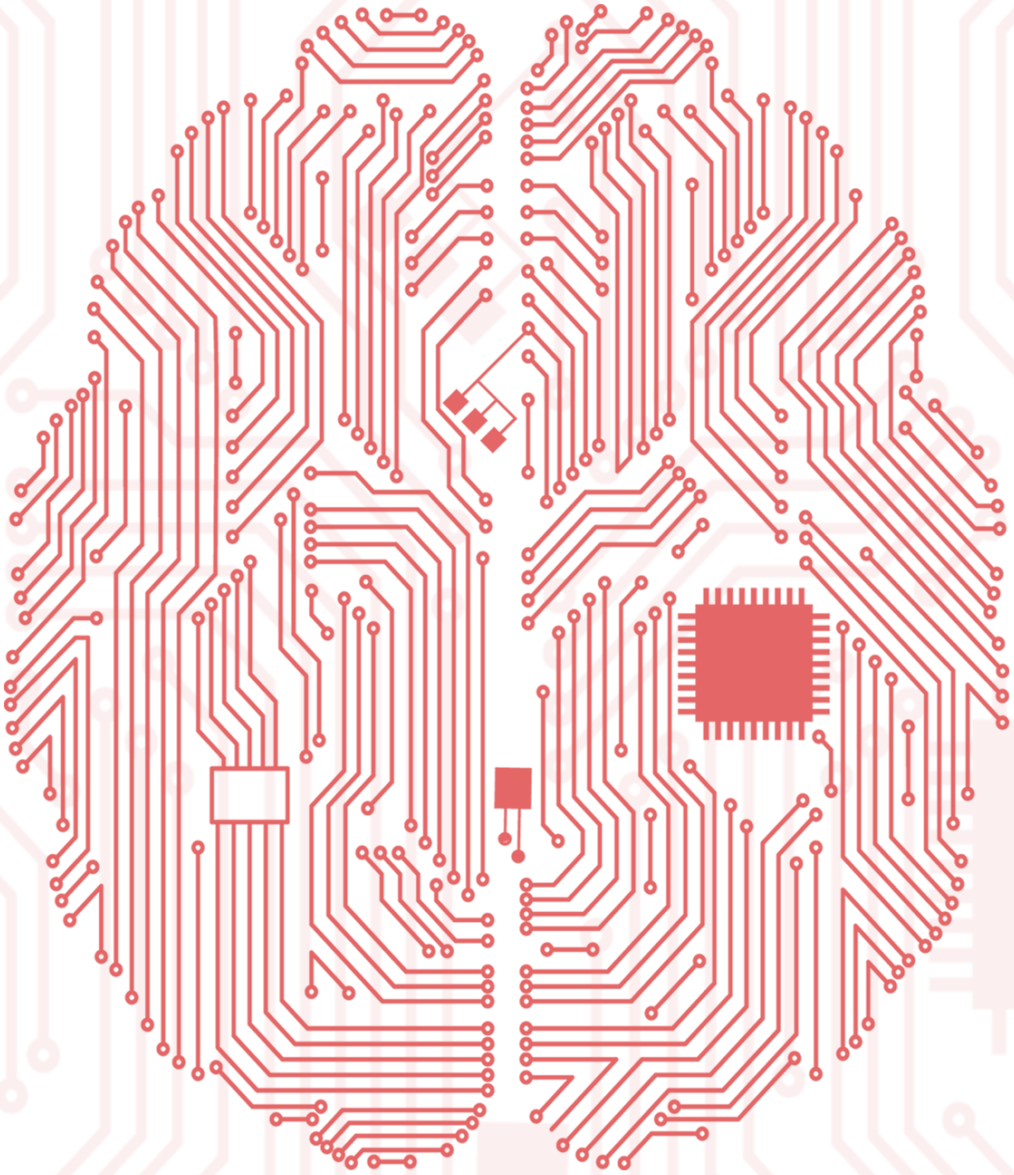
DEEP LEARNING FOR COMPUTER VISION



WITH PYTHON

Dr. Adrian Rosebrock

pyimagesearch

Deep Learning for Computer Vision with Python

Starter Bundle

Dr. Adrian Rosebrock

1st Edition (1.1.0)

PUBLISHED BY PYIMAGESEARCH

PYIMAGESEARCH.COM

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.

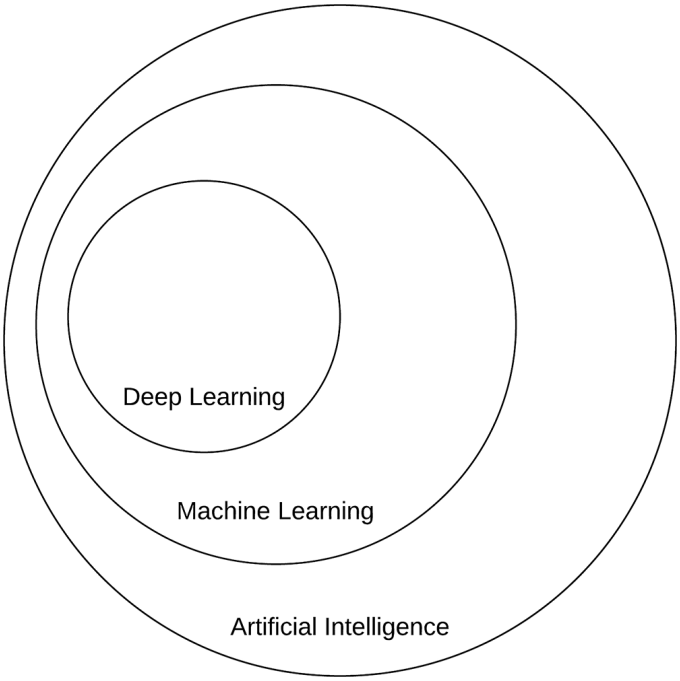


Figure 2.1: A Venn diagram describing deep learning as a subfield of machine learning which is in turn a subfield of artificial intelligence (Image inspired by Figure 1.4 of Goodfellow et al. [10]).

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

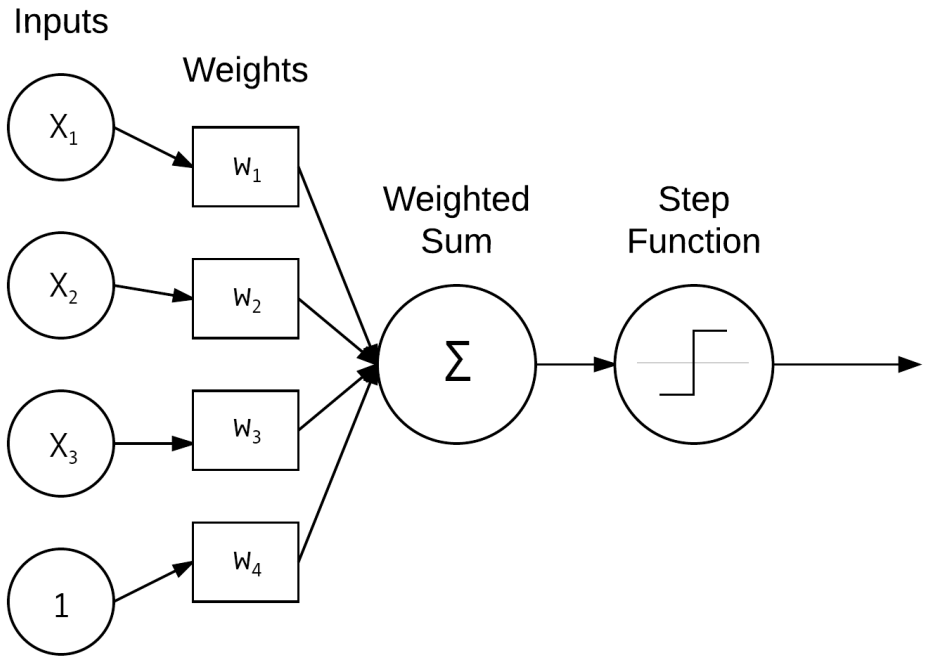


Figure 2.2: An example of the simple Perceptron network architecture that accepts a number of inputs, computes a weighted sum, and applies a step function to obtain the final prediction. We'll review the Perceptron in detail inside Chapter 10.

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-*

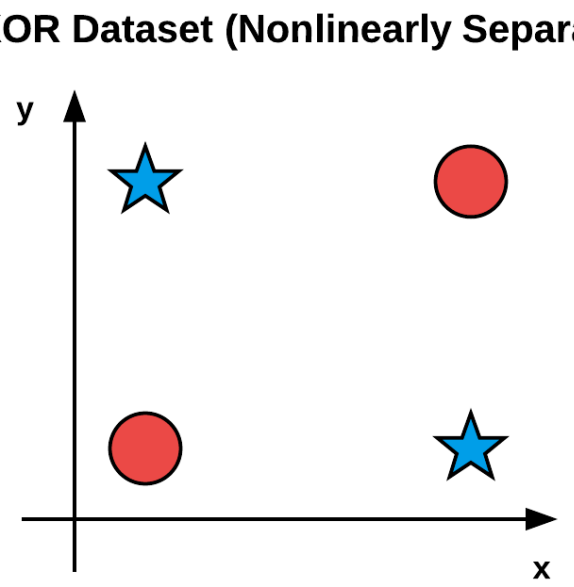


Figure 2.3: The XOR (E(X)clusive Or) dataset is an example of a nonlinear separable problem that the Perceptron *cannot* solve. Take a second to convince yourself that it is impossible to draw a single line that separates the blue stars from the red circles.

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

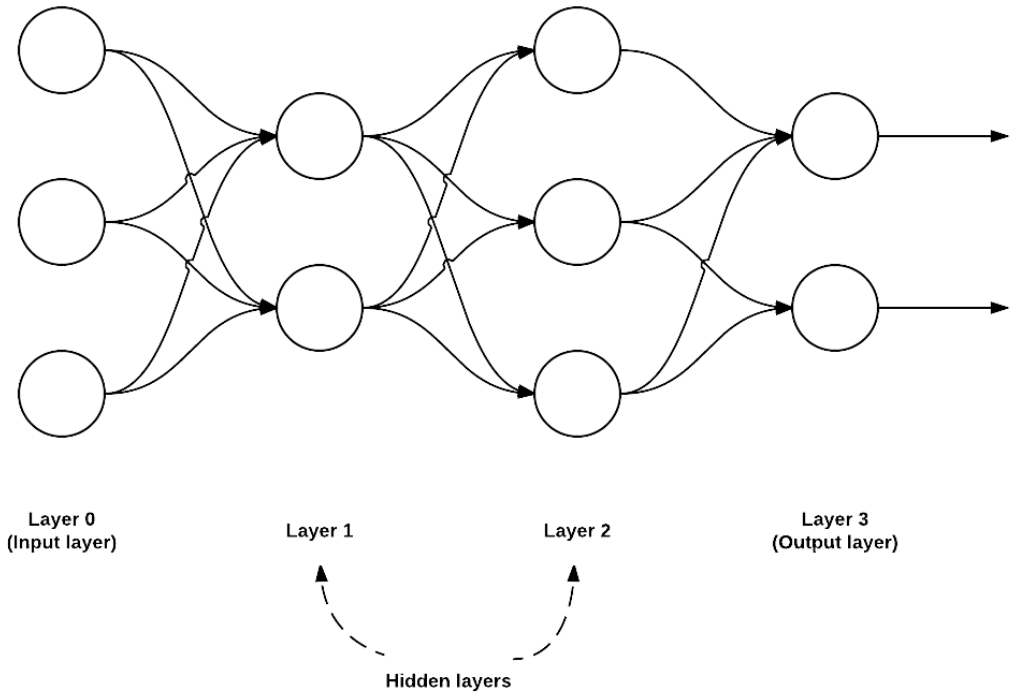


Figure 2.4: A multi-layer, feedforward network architecture with an input layer (3 nodes), two hidden layers (2 nodes in the first layer and 3 nodes in the second layer), and an output layer (2 nodes).

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

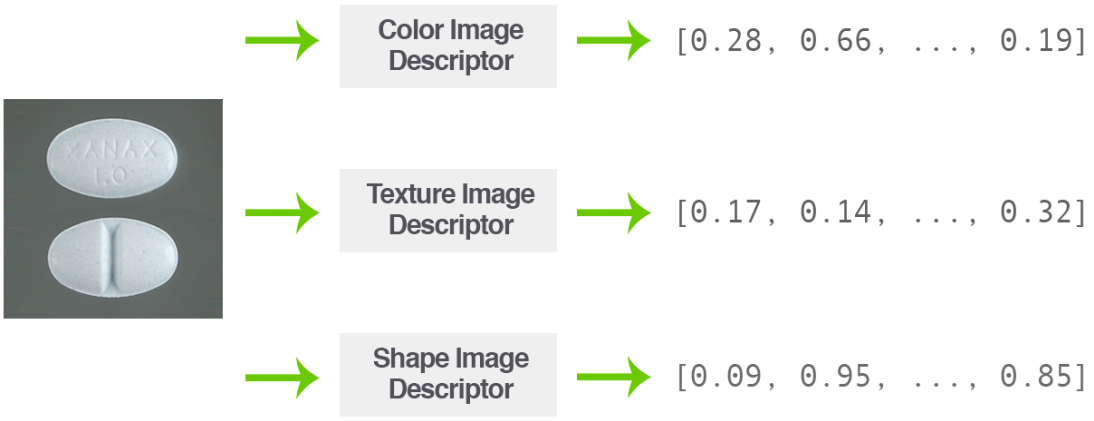


Figure 2.5: Quantifying the contents of an image containing a prescription pill medication via a series of blackbox color, texture, and shape image descriptors.

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].

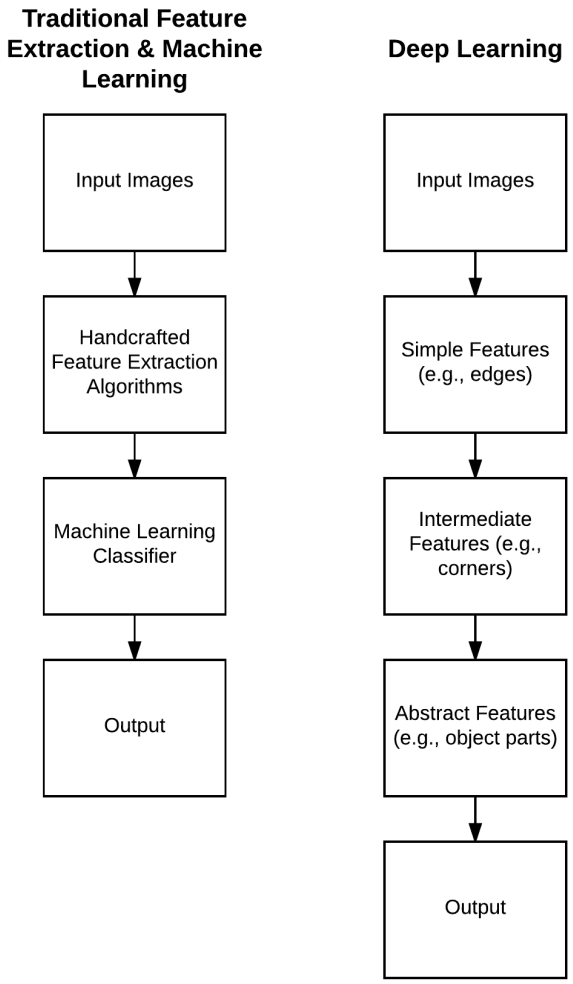


Figure 2.7: **Left:** Traditional process of taking an input set of images, applying hand-designed feature extraction algorithms, followed by training a machine learning classifier on the features. **Right:** Deep learning approach of stacking layers on top of each other that *automatically* learn more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

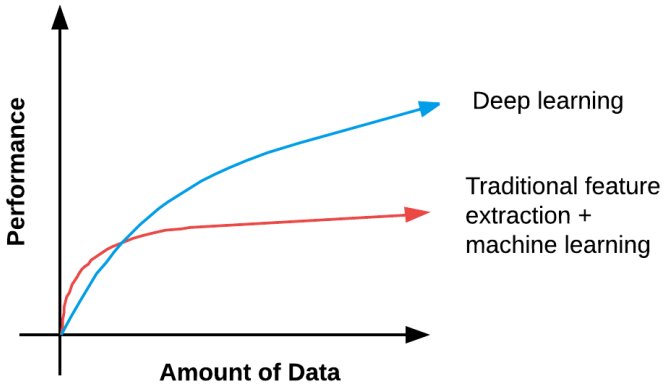


Figure 2.8: As the amount of data available to deep learning algorithms increases, accuracy does as well, substantially outperforming traditional feature extraction + machine learning approaches.

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.

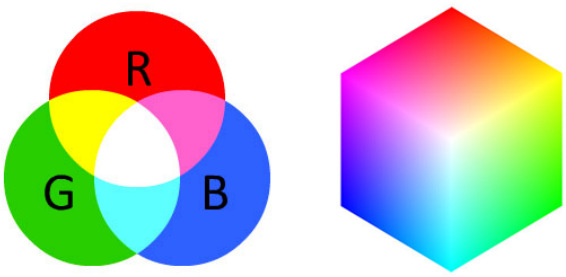


Figure 3.3: **Left:** The RGB color space is *additive*. The more red, green and blue you mix together, the closer you get to *white*. **Right:** The RGB cube.

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.

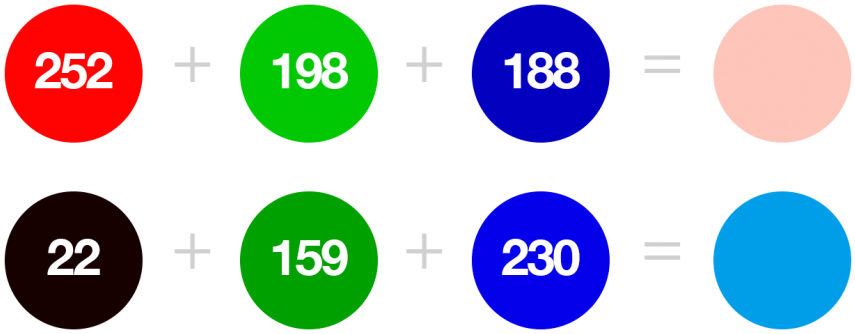


Figure 3.4: **Top:** An example of adding various Red, Green, and Blue color components together to create a "caucasian flesh tone", perhaps useful in a skin detection program. **Bottom:** Creating the specific shade of blue in the "PyImageSearch" logo by mixing various amounts of Red, Green, and Blue.

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).

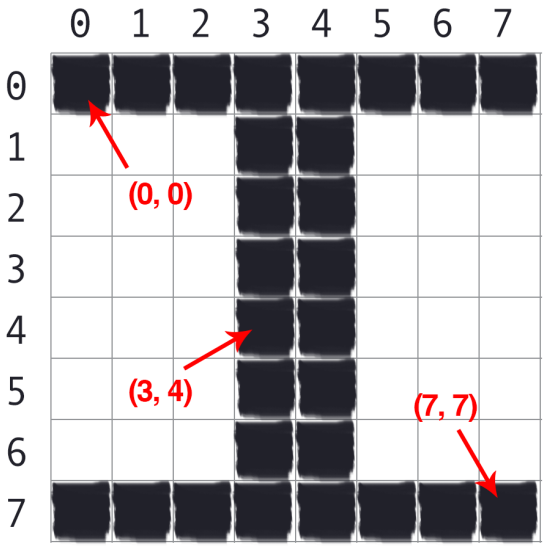


Figure 3.6: The letter "I" placed on a piece of graph paper. Pixels are accessed by their (x,y)-coordinates, where we go x columns to the right and y rows down, keeping in mind that Python is zero-indexed.

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

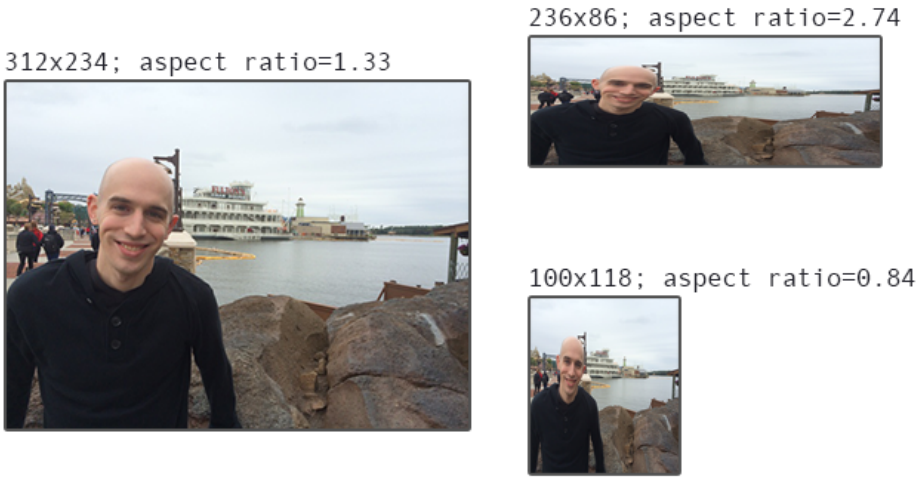


Figure 3.8: **Left:** Original image. **Top and Bottom:** Resulting distorted images after resizing without preserving the aspect ratio (i.e., the ratio of the width to the height of the image).

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.

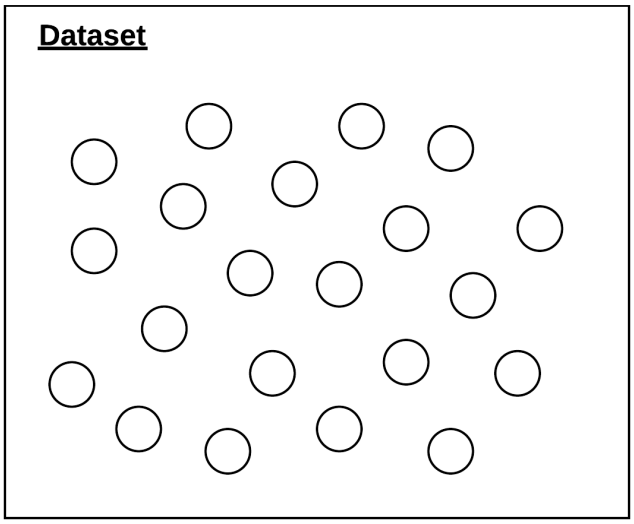


Figure 4.2: A dataset (outer rectangle) is a *collection* of data points (circles).

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.

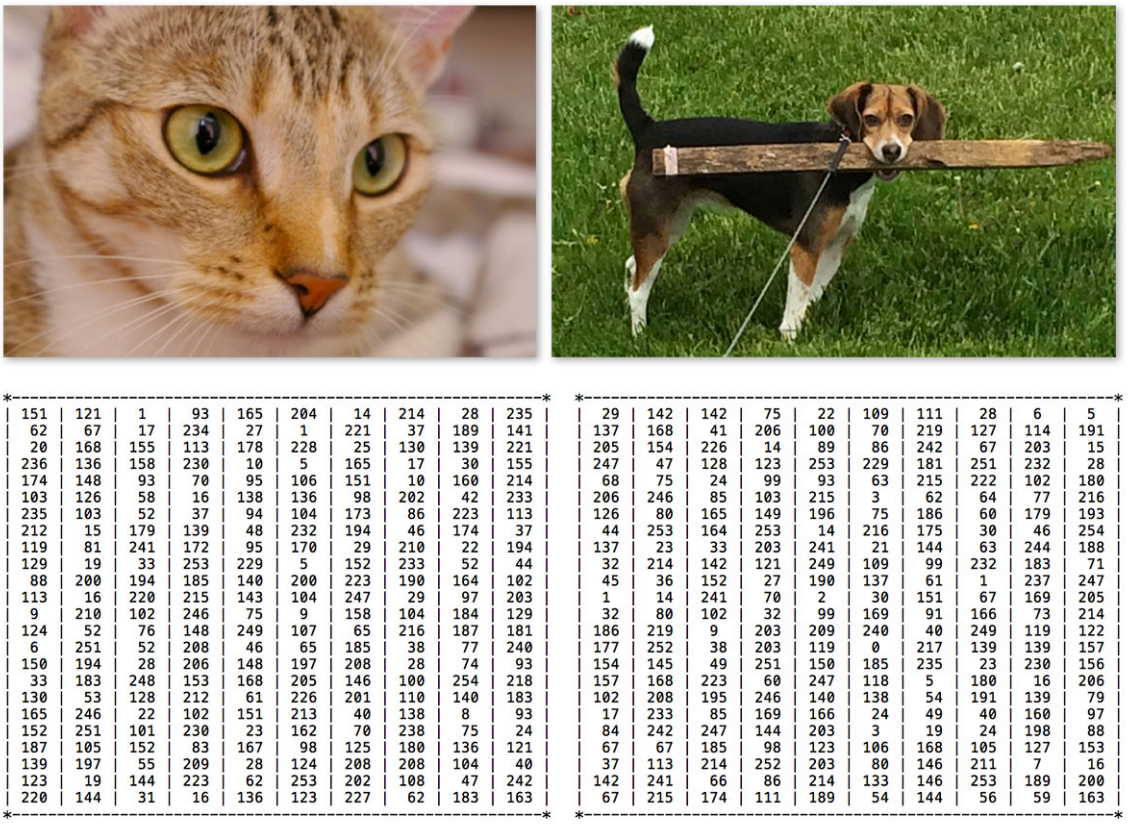
We might describe the image as follows:

- **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:

```
>>> fib(13)
233
```

And finally the 35th number:

```
>>> fib(35)
9227465
```

As you can see, the Fibonacci sequence is straightforward and is an example of a family of functions that:

1. Accepts an input, returns an output.
2. The process is well defined.
3. The output is easily verifiable for correctness.
4. Lends itself well to code coverage and test suites.

In general, you've probably written thousands upon thousands of procedural functions like these in your life. Whether you're computing a Fibonacci sequence, pulling data from a database, or calculating the mean and standard deviation from a list of numbers, these functions are all well defined and easily verifiable for correctness.

Unfortunately, this is not the case for deep learning and image classification!

Recall from Section 4.1.2 where we looked at the pictures of a cat and a dog, replicated in Figure 4.6 for convenience. Now, imagine trying to write a procedural function that can not only tell the difference between *these two photos*, but *any* photo of a cat and a dog. How would you go about accomplishing this task? Would you check individual pixel values at various (x, y)-coordinates? Write hundreds of `if/else` statements? And how would you maintain and verify the correctness of such as massive rule-based system? The short answer is: **you don't**.

Figure 4.6: How might you go about writing a piece of software to recognize the difference between dogs and cats in images? Would you inspect individual pixel values? Take a rule-based approach? Try to write (and maintain) hundreds of *if/else* statements?

Unlike coding up an algorithm to compute the Fibonacci sequence or sort a list of numbers, it's not intuitive or obvious how to create an algorithm to tell the difference between pictures of cats and dogs. Therefore, instead of trying to construct a rule-based system to describe what each category "looks like", we can instead take a *data driven approach* by supplying *examples* of what each category looks like and then *teach* our algorithm to recognize the difference between the categories using these examples.

We call these examples our *training dataset* of labeled images, where each data point in our training dataset consists of:

1. An image

2. The label/category (i.e., dog, cat, panda, etc.) of the image

Again, it's important that each of these images have labels associated with them because our supervised learning algorithm will need to see these labels to "teach itself" how to recognize each category. Keeping this in mind, let's go ahead and work through the four steps to constructing a deep learning model.

4.3.2 Step #1: Gather Your Dataset

The first component of building a deep learning network is to gather our initial dataset. We need the *images themselves* as well as the *labels* associated with each image. These labels should come from a finite set of categories, such as: `categories = dog, cat, panda`.

Furthermore, the number of images for each category should be approximately uniform (i.e., the same number of examples per category). If we have twice the number of cat images than dog images, and five times the number of panda images than cat images, then our classifier will become naturally biased to overfitting into these heavily-represented categories.

Class imbalance is a common problem in machine learning and there exist a number of ways to overcome it. We'll discuss some of these methods later in this book, but keep in mind the best method to avoid learning problems due to class imbalance is to simply avoid class imbalance entirely.

4.3.3 Step #2: Split Your Dataset

Now that we have our initial dataset, we need to split it into two parts:

1. A *training set*
2. A *testing set*

A *training set* is used by our classifier to "learn" what each category looks like by making predictions on the input data and then correct itself when predictions are wrong. After the classifier has been trained, we can evaluate the performing on a *testing set*.

It's extremely important that the training set and testing set are *independent of each other* and *do not overlap!* If you use your testing set as part of your training data, then your classifier has an unfair advantage since it has already seen the testing examples before and "learned" from them. Instead, you must keep this testing set entirely separate from your training process and use it ***only to evaluate your network***.

Common split sizes for training and testing sets include $66.\overline{6}\%33.\overline{3}\%$, $75\%/25\%$, and $90\%/10\%$, respectively. (Figure 4.7):

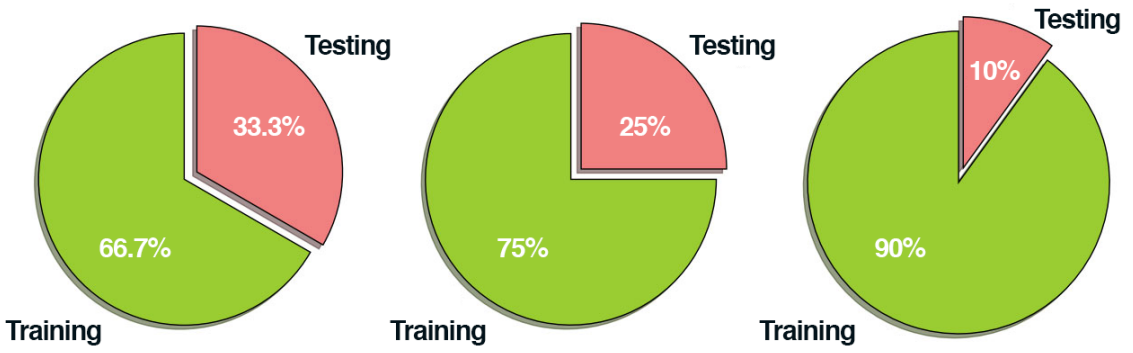


Figure 4.7: Examples of common training and testing data splits.

These data splits make sense, **but what if you have parameters to tune?** Neural networks have a number of knobs and levers (ex., learning rate, decay, regularization, etc.) that need to be tuned

and dialed to obtain optimal performance. We'll call these types of parameters *hyperparameters*, and it's *critical* that they get set properly.

In practice, we need to test a bunch of these hyperparameters and identify the set of parameters that works the best. You might be tempted to use your testing data to tweak these values, ***but again, this is a major no-no!*** The test set is *only* used in evaluating the performance of your network.

Instead, you should create a *third* data split called the **validation set**. This set of the data (normally) comes from the training data and is used as "fake test data" so we can tune our hyperparameters. Only after have we determined the hyperparameter values using the validation set do we move on to collecting final accuracy results in the testing data.

We normally allocate roughly 10-20% of the training data for validation. If splitting your data into chunks sounds complicated, it's actually not. As we'll see in our next chapter, it's quite simple and can be accomplished with only a **single line of code** thanks to the scikit-learn library.

4.3.4 Step #3: Train Your Network

Given our training set of images, we can now train our network. The goal here is for our network to learn how to recognize each of the categories in our labeled data. When the model makes a mistake, it learns from this mistake and improves itself.

So, how does the actual "learning" work? In general, we apply a form of gradient descent, as discussed in Chapter 9. The remainder of this book is dedicated to demonstrating how to train neural networks from scratch, so we'll defer a detailed discussion of the training process until then.

4.3.5 Step #4: Evaluate

Last, we need to evaluate our trained network. For each of the images in our testing set, we present them to the network and ask it to *predict* what it thinks the label of the image is. We then tabulate the predictions of the model for an image in the testing set.

Finally, these *model predictions* are compared to the *ground-truth* labels from our testing set. The ground-truth labels represent what the image category *actually is*. From there, we can compute the number of predictions our classifier got correct and compute aggregate reports such as precision, recall, and f-measure, which are used to quantify the performance of our network as a whole.

4.3.6 Feature-based Learning versus Deep Learning for Image Classification

In the traditional, feature-based approach to image classification, there is actually a step inserted between Step #2 and Step #3 – this step is ***feature extraction***. During this phase, we apply hand-engineered algorithms such as HOG [32], LBPs [21], etc. to quantify the contents of an image based on a particular component of the image we want to encode (i.e., shape, color, texture). Given these features, we then proceed to train our classifier and evaluate it.

When building Convolutional Neural Networks, we can actually *skip* the feature extraction step. The reason for this is because CNNs are *end-to-end* models. We present the raw input data (pixels) to the network. The network then *learns* filters inside its hidden layers that can be used to discriminate amongst object classes. The output of the network is then a probability distribution over class labels.

One of the exciting aspects of using CNNs is that we no longer need to fuss over hand-engineered features – we can let our network learn the features instead. However, this tradeoff does come at a cost. Training CNNs can be a non-trivial process, so be prepared to spend considerable time familiarizing yourself with the experience and running many experiments to determine what does and does not work.

4.3.7 What Happens When my Predictions Are Incorrect?

Inevitably, you will train a deep learning network on your training set, evaluate it on your test set (finding that it obtains high accuracy), and then apply it to images that are *outside* both your training and testing set – *only to find that the network performs poorly*.

This problem is called **generalization**, the ability for a network to *generalize* and correctly predict the class label of an image that does not exist as part of its training or testing data.

The ability for a network to generalize is quite literally the most important aspect of deep learning research – if we can train networks that can generalize to outside datasets without re-training or fine-tuning, we'll make great strides in machine learning, enabling networks to be re-used in a variety of domains. The ability of a network to generalize will be discussed many times in this book, but I wanted to bring up the topic now since you will inevitably run into generalization issues, especially as you learn the ropes of deep learning.

Instead of becoming frustrated with your model not correctly classifying an image, consider the set of factors of variation mentioned above. Does your training dataset accurately reflect examples of these factors of variation? If not, you'll need to gather more training data (and read the rest of this book to learn other techniques to combat generalization).

4.4 Summary

Inside this chapter we learned what image classification is and why it's such a challenging task for computers to perform well on (even though humans do it intuitively with seemingly no effort). We then discussed the three main types of machine learning, supervised learning, unsupervised learning, semi-supervised learning – this book primarily focuses on supervised learning where we have *both* the training examples *and* the class labels associated with them. Semi-supervised learning and unsupervised learning are both open areas of research for deep learning (and in machine learning in general).

Finally, we reviewed the four steps in the deep learning classification pipeline. These steps including gathering your dataset, splitting your data into training, testing, and validation steps, training your network, and finally evaluating your model.

Unlike traditional feature-based approaches which require us to utilize hand-crafted algorithms to extract features from an image, image classification models, such as Convolutional Neural Networks, are end-to-end classifiers which internally learn features that can be used to discriminate amongst image classes.

5. Datasets for Image Classification

At this point we have accustomed ourselves to the fundamentals of the image classification pipeline – but before we dive into any code looking at actually *how* to take a dataset and build an image classifier, let's first review datasets that you'll see inside *Deep Learning for Computer Vision with Python*.

Some of these datasets are essentially "solved", enabling us to obtain extremely high-accuracy classifiers ($> 95\%$ accuracy) with little effort. Other datasets represent categories of computer vision and deep learning problems are still open research topics today and are far from solved. Finally, a few of the datasets are part of image classification competitions and challenges (e.x., Kaggle Dogs vs. Cats and cs231n Tiny ImageNet 200).

It's important to review these datasets now so that we have a high-level understanding of the challenges we can expect when working with them in later chapters.

5.1 MNIST

Figure 5.1: A sample of the MNIST dataset. The goal of this dataset is to correctly classify the handwritten digits, $0-9$.

The MNIST ("NIST" stands for *National Institute of Standards and Technology* while the "M" stands for *"modified"* as the data has been preprocessed to reduce any burden on computer vision processing and focus solely on the task of digit recognition) dataset is one of the most well studied datasets in the computer vision and machine learning literature.

The goal of this dataset is to correctly classify the handwritten digits $0-9$. In many cases, this dataset is a benchmark, a standard to which machine learning algorithms are ranked. In fact,

MNIST is *so well studied* that Geoffrey Hinton described the dataset as *"the drosophila of machine learning"* [10] (a *drosophila* is a genus of fruit fly), comparing how budding biology researchers use these fruit flies as they are easily cultured en masse, have a short generation time, and mutations are easily obtained.

In the same vein, the MNIST dataset is simple dataset for early deep learning practitioners to get their "first taste" of training a neural network without too much effort (it's *very* easy to obtain $> 97\%$ classification accuracy) – training a neural network model on MNIST is very much the *"Hello, World"* equivalent in machine learning.

MNIST itself consists of 60,000 training images and 10,000 testing images. Each feature vector is 784-dim, corresponding to the 28×28 grayscale pixel intensities of the image. These grayscale pixel intensities are unsigned integers, falling into the range $[0, 255]$. All digits are placed on a black background with the foreground being white and shades of gray. Given these raw pixel intensities, our goal is to train a neural network to correctly classify the digits.

We'll be primarily using this dataset in the early chapters of the *Starter Bundle* to help us "get our feet wet" and learn the ropes of neural networks.

5.2 Animals: Dogs, Cats, and Pandas

Figure 5.2: A sample of the 3-class animals dataset consisting of 1,000 images per dog, cat, and panda class respectively for a total of 3,000 images.

The purpose of this dataset is to correctly classify an image as contain a dog, cat, or panda. Containing only 3,000 images, the Animals dataset is meant to be another "introductory" dataset that we can quickly train a deep learning model on either our CPU or GPU and obtain reasonable accuracy.

In Chapter 10 we'll use this dataset to demonstrate how using the pixels of an image as a feature vector does not translate to a high-quality machine learning model *unless* we employ the usage of a Convolutional Neural Network (CNN).

Images for this dataset were gathered by sampling the Kaggle Dogs vs. Cats images along with the ImageNet dataset for panda examples. The Animals dataset is primarily used in the *Starter Bundle* only.

5.3 CIFAR-10

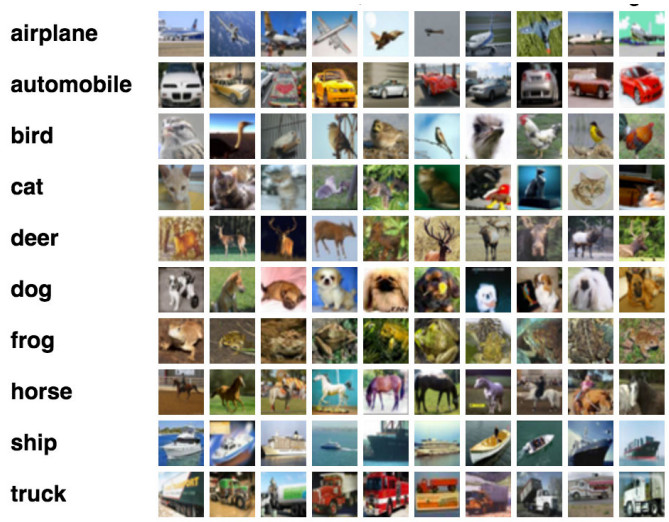


Figure 5.3: Example images from the ten class CIFAR-10 dataset.

Just like MNIST, CIFAR-10 is considered another standard benchmark dataset for image classification in the computer vision and machine learning literature. CIFAR-10 consists of 60,000 $32 \times 32 \times 3$ (RGB) images resulting in a feature vector dimensionality of 3072.

As the name suggests, CIFAR-10 consists of 10 classes, including: *airplanes*, *automobiles*, *birds*, *cats*, *deer*, *dogs*, *frogs*, *horses*, *ships*, and *trucks*.

While it's quite easy to train a model that obtains $> 97\%$ classification accuracy on MNIST, it's *substantially harder* obtain such a model for CIFAR-10 (and it's bigger brother, CIFAR-100) [50].

The challenge comes from the *dramatic variance* in how objects appear. For example, we can no longer assume that an image containing a green pixel at a given (x, y)-coordinate is a frog. This pixel could be part of the background of a forest that contains a deer. Or, the pixel could simply be the color of a green truck.

These assumptions are a stark contrast to the MNIST dataset where the network can learn assumptions regarding the spatial distribution of pixel intensities. For example, the spatial distribution of foreground pixels of the number 1 is substantially different than a 0 or 5.

While being a small dataset, CIFAR-10 is still regularly used to benchmark new CNN architectures. We'll be using CIFAR-10 in both the *Starter Bundle* and the *Practitioner Bundle*.

5.4 SMILES

As the name suggests, the SMILES dataset [51] consists of images of faces that are either *smiling* or *not smiling*. In total, there are 13,165 grayscale images in the dataset, with each image having a size of 64×64.

Images in this dataset are *tightly cropped* around the face allowing us to devise machine learning algorithms that focus solely on the task of smile recognition. Decoupling *computer vision preprocessing* from *machine learning* (especially for benchmark datasets) is a common trend you'll

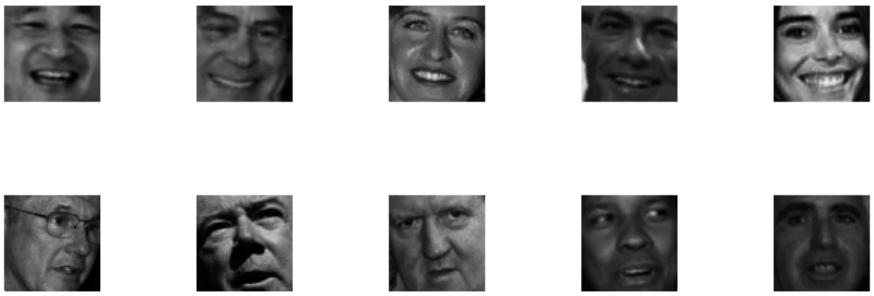


Figure 5.4: Top: Examples of "smiling" faces. Bottom: Samples of "not smiling" faces. We will later train a Convolutional Neural Network to recognize between smiling and not smiling faces in real-time video streams.

see when reviewing popular benchmark datasets. In some cases, it's unfair to assume that a machine learning researcher has enough exposure to computer vision to properly preprocess a dataset of images prior to applying their own machine learning algorithms.

That said, this trend is *quickly changing*, and any practitioner interested in applying machine learning to computer vision problems is assumed to have at least a rudimentary background in computer vision. This trend will continue in the future, so if you plan on studying deep learning for computer vision in any depth, definitely be sure to supplement your education with a bit of computer vision, even if it's just the fundamentals.

If you find that you need to improve your computer vision skills, take a look at *Practical Python and OpenCV* [8].

5.5 Kaggle: Dogs vs. Cats

The Dogs vs. Cats challenge is part of a Kaggle competition to devise a learning algorithm that can correctly classify an image as containing a *dog* or a *cat*. A total of 25,000 images are provided to train your algorithm with *varying image resolutions*. A sample of the dataset can be seen in Figure 5.5.

How you decide to preprocess your images can lead to varying performance levels, again demonstrating that a background in computer vision and image processing basics will go a long way when studying deep learning.

We'll be using this dataset in the *Practitioner Bundle* when I demonstrate how to claim a top-25 position on the Kaggle Dogs vs. Cats leaderboard using the AlexNet architecture.

5.6 Flowers-17

The Flowers-17 dataset is a 17 category dataset with 80 images per class curated by Nilsback et al. [52]. The goal of this dataset is to correctly predict the species of flower for a given input image. A sample of the Flowers-17 dataset can be seen in Figure 5.6.

Flowers-17 can be considered a challenging dataset due to the dramatic changes in scale, viewpoint angles, background clutter, varying lighting conditions, and intra-class variation. Furthermore, with only 80 images per class, it becomes challenging for deep learning models to learn a representation for each class *without* overfitting. As a general rule of thumb, it's advisable to have 1,000-5,000 example images *per class* when training a deep neural network [10].

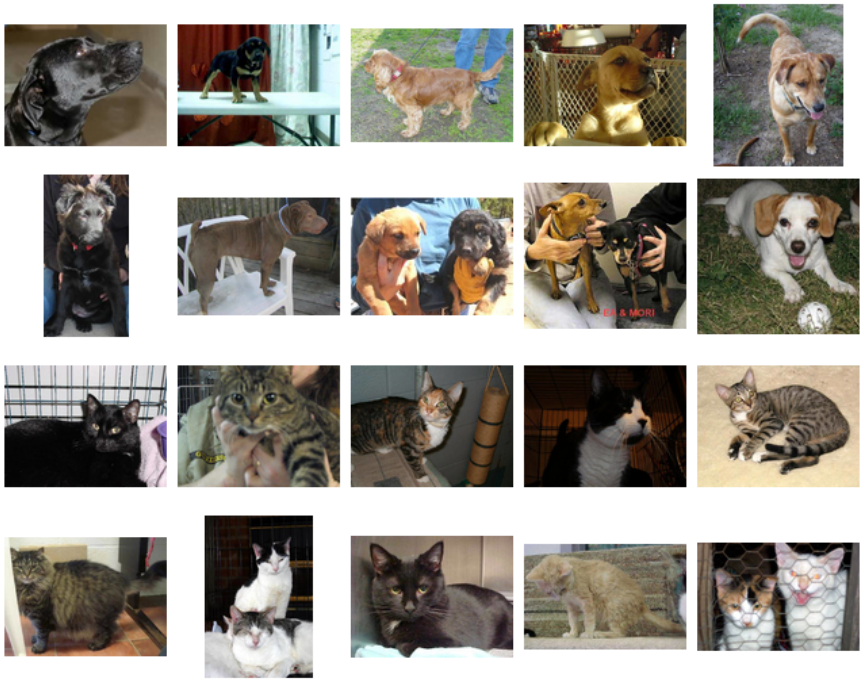


Figure 5.5: Samples from the Kaggle Dogs vs. Cats competition. The goal of this 2-class challenge is to correctly identify a given input image as containing a "dog" or "cat".

We will study the Flowers-17 dataset inside the *Practitioner Bundle* and explore methods to improve classification using transfer learning methods such as feature extraction and fine-tuning.

5.7 CALTECH-101

Introduced by Fei-Fei et al. [53] in 2004, the CALTECH-101 dataset is a popular benchmark dataset for object detection. Typically used for *object detection* (i.e., predicting the (x, y)-coordinates of the bounding box for a particular object in an image), we can use CALTECH-101 to study deep learning algorithms as well.

The dataset of 8,677 images includes 101 categories spanning a diverse range of objects, including elephants, bicycles, soccer balls, and even human brains, just to name a few. The CALTECH-101 dataset exhibits *heavy* class imbalances (meaning that there are more example images for some categories than others), making it interesting to study from a class imbalance perspective.

Previous approaches to classifying images to CALTECH-101 obtained accuracies in the range of 35-65% [54, 55, 56]. However, as I'll demonstrate in the *Practitioner Bundle*, **it's easy for us to leverage deep learning for image classification to obtain over 99% classification accuracy**.

5.8 Tiny ImageNet 200

Stanford's excellent *cs231n: Convolutional Neural Networks for Visual Recognition* class [57] has put together an image classification challenge for students similar to the ImageNet challenge, but smaller in scope. There are a total of 200 image classes in this dataset with 500 images for training, 50 images for validation, and 50 images for testing per class. Each image has been preprocessed and cropped to $64 \times 64 \times 3$ pixels making it easier for students to focus on deep learning techniques rather than computer vision preprocessing functions.

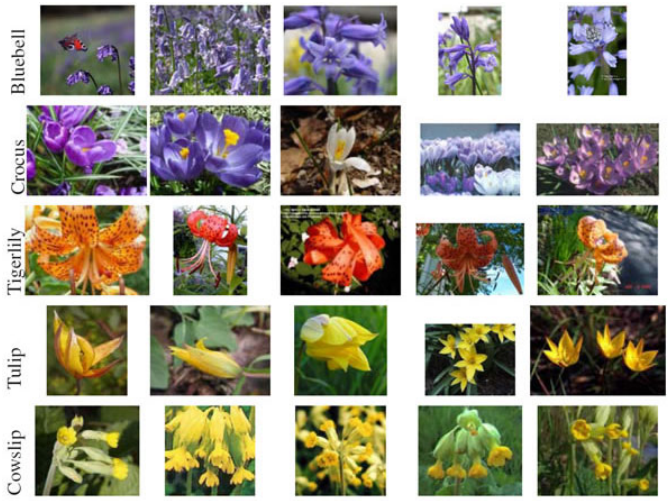


Figure 5.6: A sample of five (out of the seventeen total) classes in the Flowers-17 dataset where each class represents a *specific* flower species.

However, as we'll find in the *Practitioner Bundle*, the preprocessing steps applied by Karpathy and Johnson actually make the problem a bit *harder* as some of the important, discriminating information is cropped out during the preprocessing task. That said, I'll be demonstrating how to train the VGGNet, GoogLenet, and ResNet architectures on this dataset and claim a top position on the leaderboard.

5.9 Adience

The Adience dataset, constructed by Eidinger et al. 2014 [58], is used to facilitate the study of age and gender recognition. A total of 26,580 images are included in the dataset with ages ranging from 0-60. The goal of this dataset is to correctly predict *both* the age and gender of the subject in the image. We'll discuss the Adience dataset further (and build our own age and gender recognition systems) inside the *ImageNet Bundle*. You can see a sample of the Adience dataset in Figure 5.7.

5.10 ImageNet

Within the computer vision and deep learning communities, you might run into a bit of contextual confusion regarding what ImageNet is and isn't.

5.10.1 What Is ImageNet?

ImageNet is actually a *project* aimed at labeling and categorizing images into almost 22,000 categories based on a defined set of words and phrases.

At the time of this writing, there are over *14 million images* in the ImageNet project. To organize such a massive amount of data, ImageNet follows the WordNet hierarchy [59]. Each meaningful word/phrase inside WordNet is called a "synonym set" or "synset" for short. Within the ImageNet project, images are organized according to these synsets, with the goal being to have 1,000+ images per synset.

5.10.2 ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

In the context of computer vision and deep learning, whenever you hear people talking about ImageNet, they are very likely referring to the *ImageNet Large Scale Visual Recognition Challenge* [42],

Figure 5.7: A sample of the Adience dataset for age and gender recognition. Age ranges span from 0-60+.

or simply ILSVRC for short.

The goal of the image classification track in this challenge is to train a model that can classify an image into *1,000 separate categories* using approximately 1.2 million images for training, 50,000 for validation, and 100,000 for testing. These 1,000 image categories represent object classes that we encounter in our day-to-day lives, such as species of dogs, cats, various household objects, vehicle types, and much more. You can find the full list of object categories in the ILSVRC challenge here: http://pyimg.co/x1ler.

When it comes to image classification, the ImageNet challenge is the *de facto* standard for computer vision classification algorithms – and the leaderboard for this challenge has been *dominated* by Convolutional Neural Networks and deep learning techniques since 2012.

Inside the *ImageNet Bundle* I'll be demonstrating how to train seminal network architectures (AlexNet, SqueezeNet, VGGNet, GoogLeNet, ResNet) *from scratch* on this popular dataset, allowing you to replicate the state-of-the-art results you see in the respective research papers.

5.11 Kaggle: Facial Expression Recognition Challenge

Another challenge put together by Kaggle, the goal of the Facial Expression Recognition Challenge (FER) is to correctly identify the *emotion* a person is experiencing simply from a picture of their face. A total of 35,888 images are provided in the FER challenge with the goal to label a given facial expression into seven different categories:

1. Angry
2. Disgust (sometimes grouped in with "Fear" due to class imbalance)
3. Fear
4. Happy
5. Sad
6. Surprise

Figure 5.8: A collage of ImageNet examples put together by Stanford University. This dataset is massive with over 1.2 million images and 1,000 possible object categories. ImageNet is considered the *de facto* standard for benchmarking image classification algorithms.

7. Neutral

I'll be demonstrating how to use this dataset for emotion recognition inside the *ImageNet Bundle*.

5.12 Indoor CVPR

The Indoor Scene Recognition dataset [60], as the name suggests, consists of a number of indoor scenes, including stores, houses, leisure spaces, working areas, and public spaces. The goal of this dataset is to correctly train a model that can recognize each of the areas. However, instead of using this dataset for its original intended purpose, we'll instead be using it inside the *ImageNet Bundle* to automatically **detect** and **correct** image orientation.

5.13 Stanford Cars

Another dataset put together by Stanford, the Cars Dataset [61] consists of 16,185 images of 196 classes of cars. You can slice-and-dice this dataset any way you wish based on vehicle make, model, or even manufacturer year. Even though there are relatively few images per class (with heavily class imbalances), I'll demonstrate how to use Convolutional Neural Networks to obtain $> 95\%$ **classification accuracy** when labeling the make and model of a vehicle.

5.14 Summary

In this chapter, we reviewed the datasets you'll encounter in the remainder of *Deep Learning for Computer Vision with Python*. Some of these datasets are considered "toy" datasets, small sets of images that we can use to learn the ropes of neural networks and deep learning. Other datasets are popular due to historical reasons and serve as excellent benchmarks to evaluate new model architectures. Finally, datasets such as ImageNet are still open-ended research topics and are used to advance the state-of-the-art for deep learning.

Take the time to briefly familiarize yourself with these datasets now – I'll be discussing each of the datasets in detail when they are first introduced in their respective chapters.

Figure 5.9: A sample of the facial expressions inside the Kaggle: Facial Expression Recognition Challenge. We will train a CNN to recognize and identify each of these emotions. This CNN will also be able to run in *real-time* on your CPU, enabling you to recognize emotions in video streams.

Figure 5.10: The Stanford Cars Dataset consists of 16,185 images with 196 vehicle make and model classes. We'll learn how obtain $> 95\%$ classification accuracy on this dataset inside the *ImageNet Bundle*.

6. Configuring Your Development Environment

When it comes to learning a new technology (especially deep learning), configuring your development environment tends to be half the battle. Between different operating systems, varying dependency versions, and the actual libraries themselves, configuring your own deep learning development environment can be quite the headache.

These issues are all *further compounded* by the speed in which deep learning libraries are updated and released – new features push innovation, but also break previous versions. The CUDA Toolkit, in particular, is a great example: on average there are 2-3 *new releases* of CUDA *every year*.

With each new release brings optimizations, new features, and the ability to train neural networks faster. But each release further complicates backward compatibility. This fast release cycle implies that deep learning is not only dependent on *how* you configured your development environment but *when* you configured it as well. **Depending on the timeframe, *your environment may be obsolete!***

Due to the rapidly changing nature of deep learning dependencies and libraries, I've decided to move much of this chapter to the **Companion Website** (http://dl4cv.pyimagesearch.com/) so that new, fresh tutorials will *always* be available for you to use.

You should use this chapter to help familiarize yourself with the various deep learning libraries we'll be using in this book, then follow instructions on the pages that link to those libraries from this book.

6.1 Libraries and Packages

In order to become a successful deep learning practitioner, we need the right set of tools and packages. This section details the programming language along with the primary libraries we'll be using to study deep learning for computer vision.

6.1.1 Python

We'll be utilizing the Python programming language for all examples inside *Deep Learning for Computer Vision with Python*. Python is an easy language to learn and is hands-down the **best**

way to work with deep learning algorithms. The simple, intuitive syntax allows you to focus on learning the basics of deep learning, rather than spending hours fixing crazy compiler errors in other languages.

6.1.2 Keras

To build and train our deep learning networks we'll primarily be using the Keras library. Keras supports *both* TensorFlow and Theano, making it *super easy* to build and train networks quickly. Please refer to Section 6.2 for more information on TensorFlow and Theano compatibility with Keras.

6.1.3 Mxnet

We'll also be using mxnet, a deep learning library that specializes in *distributed, multi-machine learning*. The ability to parallelize training across multiple GPUs/devices is *critical* when training deep neural network architectures on massive image datasets (such as ImageNet).

The mxnet library is only used in the *ImageNet Bundle* of this book.

6.1.4 OpenCV, scikit-image, scikit-learn, and more

Since this book focuses on applying deep learning to *computer vision*, we'll be leveraging a few extra libraries as well. You *do not* need to be an expert in these libraries or have prior experience with them to be successful when using this book, but I *do* suggest familiarizing yourself with the basics of OpenCV if you can. The first five chapters of *Practical Python and OpenCV* are more than sufficient to understand the basics of the OpenCV library.

The main goal of OpenCV is real-time image processing. This library has been around since 1999, but it wasn't until the 2.0 release in 2009 that we saw the incredible Python support which included representing images as NumPy arrays.

OpenCV itself is written in C/C++, but Python bindings are provided when running the install. OpenCV is *hands down* the *de-facto standard* when it comes to image processing, so we'll make use of it when loading images from disk, displaying them to our screen, and performing basic image processing operations.

To complement OpenCV, we'll also be using a tiny bit of scikit-image [62] (scikit-image.org), a collection of algorithms for image processing.

Scikit-learn [5] (scikit-learn.org), is an open-source Python library for machine learning, cross-validation, and visualization – this library complements Keras well and helps us from not "reinventing the wheel", especially when it comes to creating training/testing/validation splits and validating the accuracy of our deep learning models.

6.2 Configuring Your Development Environment?

If you're ready to configure your deep learning environment, just click the link below and follow the provided instructions for your operating system and whether or not you will be using a GPU:

http://pyimg.co/k81c6

If you have not already created your account on the companion website for *Deep Learning for Computer Vision with Python*, please see the first few pages of this book (immediately following the Table of Contents) for the registration link. From there, create your account and you'll be able to access the supplementary material.

6.3 Preconfigured Virtual Machine

I realize that configuring your development environment can not only be a *time consuming, tedious task*, but potentially a *major barrier to entry* if you're new to Unix-based environments. Because of this difficulty, your purchase of *Deep Learning for Computer Vision with Python* includes a preconfigured Ubuntu VirtualBox virtual machine that ships with all the necessary deep learning and computer vision libraries you'll need to be successful when using this book *preconfigured* and *pre-installed*.

Make sure you download the `VirtualMachine.zip` included with your bundle to have access to this virtual machine. Instructions on how to setup and use your virtual machine can be found inside the `README.pdf` included with your download of this book.

6.4 Cloud-based Instances

A major downside of the Ubuntu VM is that by the very definition of a virtual machine, a VM is not allowed to access the physical components of your host machine (such as a GPU). When training larger deep learning networks, having a GPU is extremely beneficial.

For those who wish to have access to a GPU when training their neural networks, I would suggest either:

1. Configuring an Amazon EC2 instance with GPU support.
2. Signing up for a FloydHub account and configure your GPU instance in the cloud.

It's important to note that each of these options charge based on the number of hours (EC2) or seconds (FloydHub) that your instance is booted for. If you decide to go the "GPU in the cloud route" be sure to compare prices and be conscious of your spending – there is nothing worse than getting a large, unexpected bill for cloud usage.

If you choose to use a cloud-based instance then I would encourage you to use my pre-configured Amazon Machine Instance (AMI). The AMI comes with all deep learning libraries you'll need in this book pre-configured and pre-installed.

To learn more about the AMI, please refer to *Deep Learning for Computer Vision with Python* companion website.

6.5 How to Structure Your Projects

Now that you've had a chance to configure your development environment, take a second now and download the `.zip` of the code and datasets associated with the *Starter Bundle*.

After you've downloaded the file, unarchive it, and you'll see the following directory structure:

```
|--- sb_code
|    |--- chapter07-first_image_classifier
|    |--- chapter08-parameterized_learning
|    |--- chapter09-optimization_methods
...
|    |--- datasets
```

Each chapter (that includes accompanying code) has its own directory. Each directory then includes:

- The source code for the chapter.
- The `pyimagesearch` library for deep learning that you'll be creating as you follow along with the book.
- Any additional files needed to run the respective examples.

The `datasets` directory, as the name implies, contains all image datasets for the *Starter Bundle*.

As an example, let's say I wanted to train my first image classifier. I would first change directory into `chapter07-first_image_classifier` and then execute the `knn.py` script, pointing the `--dataset` command line argument to the `animals` dataset:

```
$ cd ../chapter07-first_image_classifier/
$ python knn.py --dataset ../datasets/animals
```

This will instruct the `knn.py` script to train a simple k-Nearest Neighbor (k-NN) classifier on the "animals" dataset (which is a subdirectory inside `datasets`), a small collection of dogs, cats, and pandas in images.

If you are new to the command line and how to use command line arguments, I ***highly recommend*** that you read up on command line arguments and how to use them before getting to far in this book:

http://pyimg.co/vsapz

Becoming comfortable with the command line (and how to debug errors using the terminal) is a very important skill for you to develop.

Finally, as a quick note, I wanted to mention that I prefer keeping my datasets *separate* from my source code as it:

- Keeps my project structure neat and tidy
- Allows me to reuse datasets across multiple projects

I would encourage you to adopt a similar directory structure for your own projects.

6.6 Summary

When it comes to configuring your deep learning development environment, you have a number of options. If you would prefer to work from your local machine, that's totally reasonable, but you will need to compile and install some dependencies first. If you are planning on using your CUDA-compatible GPU on your local machine, a few extra install steps will be required as well.

For readers who are new to configuring their development environment or for readers who simply want to skip the process altogether, be sure to take a look at the preconfigured Ubuntu VirtualBox virtual machine included in the download of your *Deep Learning for Computer Vision with Python* bundle.

If you would like to use a GPU but do not have one attached to your system, consider using cloud-based instances such as Amazon EC2 or FloydHub. While these services *do* incur an hourly charge for usage, they can save you money when compared to buying a GPU upfront.

Finally, please keep in mind that if you plan on doing any serious deep learning research or development, consider using a Linux environment such as Ubuntu. While deep learning work can *absolutely* be done on Windows (not recommended) or macOS (totally acceptable if you are just getting started), nearly all production-level environments for deep learning leverage Linux-based operating systems – keep this fact in mind when you are configuring your own deep learning development environment.

7. Your First Image Classifier

Over the past few chapters we've spent a reasonable amount of time discussing image fundamentals, types of learning, and even a four step pipeline we can follow when building our own image classifiers. But we have yet to build an *actual* image classifier of our own.

That's going to change in this chapter. We'll start by building a few helper utilities to facilitate preprocessing and loading images from disk. From there, we'll discuss the k-Nearest Neighbors (k-NN) classifier, your first exposure to using machine learning for image classification. In fact, this algorithm is *so simple* that it doesn't do any actual "learning" at all – yet it is still an important algorithm to review so we can appreciate how neural networks learn from data in future chapters.

Finally, we'll apply our k-NN algorithm to recognize various species of animals in images.

7.1 Working with Image Datasets

When working with image datasets, we first must consider the total size of the dataset in terms of bytes. Is our dataset large enough to fit into the available RAM on our machine? Can we load the dataset as if loading a large matrix or array? Or is the dataset so large that it exceeds our machine's memory, requiring us to "chunk" the dataset into segments and only load parts at a time?

The datasets inside the *Starter Bundle* are all small enough that we can load them into main memory without having to worry about memory management; however, the much larger datasets inside the *Practitioner Bundle* and *ImageNet Bundle* will require us to develop some clever methods to efficiently handle loading images in a way that we can train an image classifier (without running out of memory).

That said, you should always be cognizant of your dataset size before even starting to work with image classification algorithms. As we'll see throughout the rest of this chapter, taking the time to organize, preprocess, and load your dataset is a critical aspect of building an image classifier.

7.1.1 Introducing the "Animals" Dataset

The "Animals" dataset is a simple example dataset I put together to demonstrate how to train image classifiers using simple machine learning techniques as well as advanced deep learning algorithms.

Figure 7.1: A sample of the 3-class animals dataset consisting of 1,000 images per *dog*, *cat*, and *panda* class respectively for a total of 3,000 images.

Images inside the Animals dataset belong to three distinct classes: **dogs**, **cats**, and **pandas**, with 1,000 example images per class. The dog and cat images were sampled from the Kaggle Dogs vs. Cats challenge (http://pyimg.co/ogx37) while the panda images were sampled from the ImageNet dataset [42].

Containing only 3,000 images, the Animals dataset can easily fit into the main memory of our machines, which will make training our models much faster, without requiring us to write any "overhead code" to manage a dataset that could not otherwise fit into memory. Best of all, a deep learning model can quickly be trained on this dataset on *either* a CPU or GPU. Regardless of your hardware setup, you can use this dataset to learn the basics of machine learning and deep learning.

Our goal in this chapter is to leverage the k-NN classifier to attempt to recognize each of these species in an image using only the *raw pixel intensities* (i.e., no feature extraction is taking place). As we'll see, raw pixel intensities do not lend themselves well to the k-NN algorithm. Nonetheless, this is an important benchmark experiment to run so we can appreciate why Convolutional Neural Networks are able to obtain such high accuracy on raw pixel intensities while traditional machine learning algorithms fail to do so.

7.1.2 The Start to Our Deep Learning Toolkit

As I mentioned in Section 1.5, we'll be building our own custom deep learning toolkit throughout the entirety of this book. We'll start with basic helper functions and classes to preprocess images and load small datasets, eventually building up to implementations of current state-of-the-art Convolutional Neural Networks.

In fact, this is the *exact same* toolkit I use when performing deep learning experiments of my own. This toolkit will be built piece by piece, chapter by chapter, allowing you to see the individual components that make up the package, eventually becoming a full-fledged library that can be used to rapidly build and train your own custom deep learning networks.

Let's go ahead and start defining the project structure of our toolkit:

```
|--- pyimagesearch
```

As you can see, we have a single module named `pyimagesearch`. All code that we develop will exist inside the `pyimagesearch` module. For the purposes of this chapter, we'll need to define two submodules:

```
|--- pyimagesearch
|    |--- __init__.py
|    |--- datasets
|    |    |--- __init__.py
|    |    |--- simpledatasetloader.py
|    |--- preprocessing
|    |    |--- __init__.py
|    |    |--- simplepreprocessor.py
```

The `datasets` submodule will start our implementation of a class named `SimpleDatasetLoader`. We'll be using this class to load small image datasets from disk (that can fit into main memory), optionally preprocess each image in the dataset according to a set of functions, and then return the:

1. Images (i.e., raw pixel intensities)
2. Class label associated with each image

We then have the `preprocessing` submodule. As we'll see in later chapters, there are a number of preprocessing methods we can apply to our dataset of images to boost classification accuracy, including mean subtraction, sampling random patches, or simply resizing the image to a fixed size. In this case, our `SimplePreprocessor` class will do the latter – load an image from disk and resized it to a fixed size, ignoring aspect ratio. In the next two sections we'll implement `SimplePreprocessor` and `SimpleDatasetLoader` by hand.

While we will be reviewing the entire `pyimagesearch` module for deep learning in this book, I have purposely left explanations of the `__init__.py` files as an exercise to the reader. These files simply contain shortcut imports and are not relevant to understanding the deep learning and machine learning techniques applied to image classification. If you are new to the Python programming language, I would suggest brushing up on the basics of package imports [63] (http://pyimg.co/7w238).

7.1.3 A Basic Image Preprocessor

Machine learning algorithms such as k-NN, SVMs, and even Convolutional Neural Networks require all images in a dataset to have a fixed feature vector size. In the case of images, this requirement implies that our images must be preprocessed and scaled to have identical widths and heights.

There are a number of ways to accomplish this resizing and scaling, ranging from more advanced methods that respect the aspect ratio of the original image to the scaled image to simple methods that ignore the aspect ratio and simply squash the width and height to the required dimensions. Exactly which method you should use really depends on the complexity of your *factors of variation* (Section 4.1.3) – in some cases, ignoring the aspect ratio works just fine; in other cases, you'll want to preserve the aspect ratio.

In this chapter, we'll start with the basic solution: building an image preprocessor that resizes the image, ignoring the aspect ratio. Open up `simplepreprocessor.py` and then insert the following code:

```
1  # import the necessary packages
2  import cv2
3
4  class SimplePreprocessor:
5      def __init__(self, width, height, inter=cv2.INTER_AREA):
6          # store the target image width, height, and interpolation
7          # method used when resizing
8          self.width = width
9          self.height = height
10         self.inter = inter
11
12     def preprocess(self, image):
13         # resize the image to a fixed size, ignoring the aspect
14         # ratio
15         return cv2.resize(image, (self.width, self.height),
16             interpolation=self.inter)
```

Line 2 imports our only required package, our OpenCV bindings. We then define the constructor to the `SimpleProcessor` class on **Line 5**. The constructor requires two arguments, followed by a third optional one, each detailed below:

- `width`: The target width of our input image after resizing.
- `height`: The target height of our input image after resizing.
- `inter`: An optional parameter used to control which interpolation algorithm is used when resizing.

The `preprocess` function is defined on **Line 12** requiring a single argument – the input `image` that we want to preprocess.

Lines 15 and 16 preprocess the image by resizing it to a fixed size of `width` and `height` which we then return to the calling function.

Again, this preprocessor is by definition very basic – all we are doing is accepting an input image, resizing it to a fixed dimension, and then returning it. However, when combined with the image dataset loader in the next section, this preprocessor will allow us to quickly load and preprocess a dataset from disk, enabling us to briskly move through our image classification pipeline and move onto more important aspects, *such as training our actual classifier*.

7.1.4 Building an Image Loader

Now that our `SimplePreprocessor` is defined, let's move on to the `SimpleDatasetLoader`:

```
1  # import the necessary packages
2  import numpy as np
3  import cv2
4  import os
5
6  class SimpleDatasetLoader:
7      def __init__(self, preprocessors=None):
8          # store the image preprocessor
9          self.preprocessors = preprocessors
10
11         # if the preprocessors are None, initialize them as an
12         # empty list
13         if self.preprocessors is None:
14             self.preprocessors = []
```

Lines 2-4 import our required Python packages: NumPy for numerical processing, `cv2` for our OpenCV bindings, and `os` so we can extract the names of subdirectories in image paths.

Line 7 defines the constructor to `SimpleDatasetLoader` where we can optionally pass in a list of image preprocessors (such as `SimpleProcessor`) that can be sequentially applied to a given input image.

Specifying these `preprocessors` as a list rather than a single value is important – there will be times where we first need to resize an image to a fixed size, then perform some sort of scaling (such as mean subtraction), followed by converting the image array to a format suitable for Keras. Each of these preprocessors can be implemented *independently*, allowing us to apply them sequentially to an image in an efficient manner.

We can then move on to the `load` method, the core of the `SimpleDatasetLoader`:

```
16      def load(self, imagePaths, verbose=-1):
17          # initialize the list of features and labels
18          data = []
19          labels = []
20
21          # loop over the input images
22          for (i, imagePath) in enumerate(imagePaths):
23              # load the image and extract the class label assuming
24              # that our path has the following format:
25              # /path/to/dataset/{class}/{image}.jpg
26              image = cv2.imread(imagePath)
27              label = imagePath.split(os.path.sep)[-2]
```

Our `load` method requires a single parameter – `imagePaths`, which is a list specifying the file paths to the images in our dataset residing on disk. We can also supply a value for `verbose`. This "verbosity level" can be used to print updates to a console, allowing us to monitor how many images the `SimpleDatasetLoader` has processed.

Lines 18 and 19 initialize our `data` list (i.e., the images themselves) along with `labels`, the list of class labels for our images.

On **Line 22** we start looping over each of the input images. For each of these images, we load it from disk (**Line 26**) and extract the class label based on the file path (**Line 27**). We make the assumption that our datasets are organized on disk according to the following directory structure:

```
/dataset_name/class/image.jpg
```

The `dataset_name` can be whatever the name of the dataset is, in this case `animals`. The `class` should be the name of the class label. For our example, `class` is either `dog`, `cat`, or `panda`. Finally, `image.jpg` is the name of the actual image itself.

Based on this hierarchical directory structure, we can keep our datasets neat and organized. It is thus safe to assume that all images inside the `dog` subdirectory are examples of dogs. Similarly, we assume that all images in the `panda` directory contain examples of pandas.

Nearly every dataset that we review inside *Deep Learning for Computer Vision with Python* will follow this hierarchical directory design structure – **I *strongly encourage you* to do the same for your own projects as well.**

Now that our image is loaded from disk, we can preprocess it (if necessary):

```
29              # check to see if our preprocessors are not None
30              if self.preprocessors is not None:
```

```
31                    # loop over the preprocessors and apply each to
32                    # the image
33                    for p in self.preprocessors:
34                         image = p.preprocess(image)
35
36               # treat our processed image as a "feature vector"
37               # by updating the data list followed by the labels
38               data.append(image)
39               labels.append(label)
```

Line 30 makes a quick check to ensure that our `preprocessors` is not `None`. If the check passes, we loop over each of the `preprocessors` on **Line 33** and sequentially apply them to the image on **Line 34** – this action allows us to form a *chain of preprocessors* that can be applied to every image in a dataset.

Once the image has been preprocessed, we update the `data` and `label` lists, respectively (**Lines 39 and 39**).

Our last code block simply handles printing updates to our console and then returning a 2-tuple of the `data` and `labels` to the calling function:

```
41               # show an update every ‘verbose‘ images
42               if verbose > 0 and i > 0 and (i + 1) % verbose == 0:
43                    print("[INFO] processed {}/{}".format(i + 1,
44                         len(imagePaths)))
45
46          # return a tuple of the data and labels
47          return (np.array(data), np.array(labels))
```

As you can see, our dataset loader is simple by design; however, it affords us the ability to apply any number of image processors to *every* image in our dataset with ease. The only caveat of this dataset loader is that it assumes that all images in the dataset can fit into main memory at once.

For datasets that are too large to fit into your system's RAM, we'll need to design a more complex dataset loader – I cover these more advanced dataset loaders inside the *Practitioner Bundle*. Now that we understand how to (1) preprocess an image and (2) load a collection of images from disk, we can now move on to the image classification stage.

7.2 k-NN: A Simple Classifier

The k-Nearest Neighbor classifier is *by far* the most simple machine learning and image classification algorithm. In fact, it's *so simple* that it doesn't actually "learn" anything. Instead, this algorithm directly relies on the distance between feature vectors (which in our case, are the raw RGB pixel intensities of the images).

Simply put, the k-NN algorithm classifies unknown data points by finding the *most common class* among the ***k*** *closest examples*. Each data point in the k closest data points casts a vote, and the category with the highest number of votes wins. Or, in plain English: *"Tell me who your neighbors are, and I'll tell you who you are"* [64], as Figure 7.2 demonstrates.

In order for the k-NN algorithm to work, it makes the primary assumption that images with similar visual contents lie *close together* in an n-dimensional space. Here, we can see three categories of images, denoted as *dogs*, *cats*, and *pandas*, respectively. In this pretend example we have plotted the "fluffiness" of the animal's coat along the $x-axis$ and the "lightness" of the coat along the y-axis. Each of the animal data points are grouped relatively close together in our

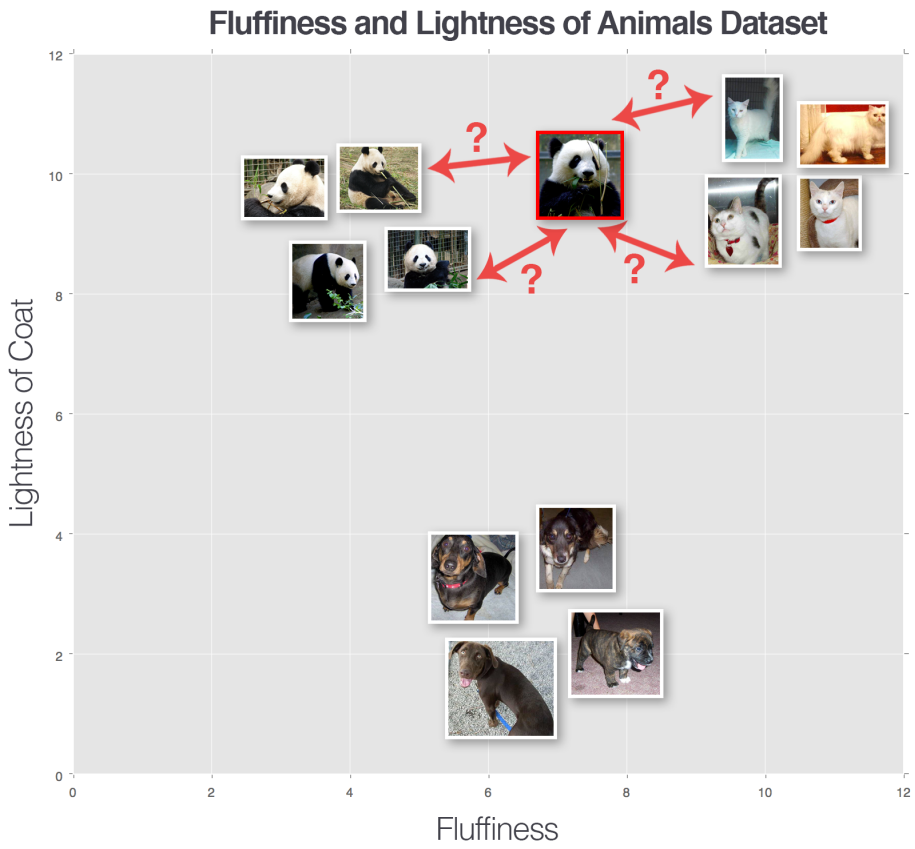


Figure 7.2: Given our dataset of dogs, cats, pandas, how might we classify the image outlined in red?

n-dimensional space. This implies that the distance between two *cat* images is *much smaller* than the distance between a *cat* and a *dog*.

However, in order to apply the k-NN classifier, we first need to select a distance metric or similarity function. A common choice includes the Euclidean distance (often called the L2-distance):

$$d(\boldsymbol{p},\boldsymbol{q}) = \sqrt{\sum_{i=1}^{N}(q_i - p_i)^2} \tag{7.1}$$

However, other distance metrics such as the Manhattan/city block (often called the L1-distance) can be used as well:

$$d(\boldsymbol{p},\boldsymbol{q}) = \sum_{i=1}^{N}|q_i - p_i| \tag{7.2}$$

In reality, you can use whichever distance metric/similarity function that most suits your data (and gives you the best classification results). However, for the remainder of this lesson, we'll be using the most popular distance metric: the Euclidean distance.

7.2.1 A Worked k-NN Example

At this point, we understand the principles of the k-NN algorithm. We know that it relies on the distance between feature vectors/images to make a classification. And we know that it requires a distance/similarity function to compute these distances.

But how do we actually *make* a classification? To answer this question, let's look at Figure 7.3. Here we have a dataset of three types of animals – *dogs*, *cats*, and *pandas* – and we have plotted them according to their *fluffiness* and *lightness of their coat*.

We have also inserted an "unknown animal" that we are trying to classify using only a **single neighbor** (i.e., $k = 1$). In this case, the nearest animal to the input image is a dog data point; thus our input image should be classified as *dog*.

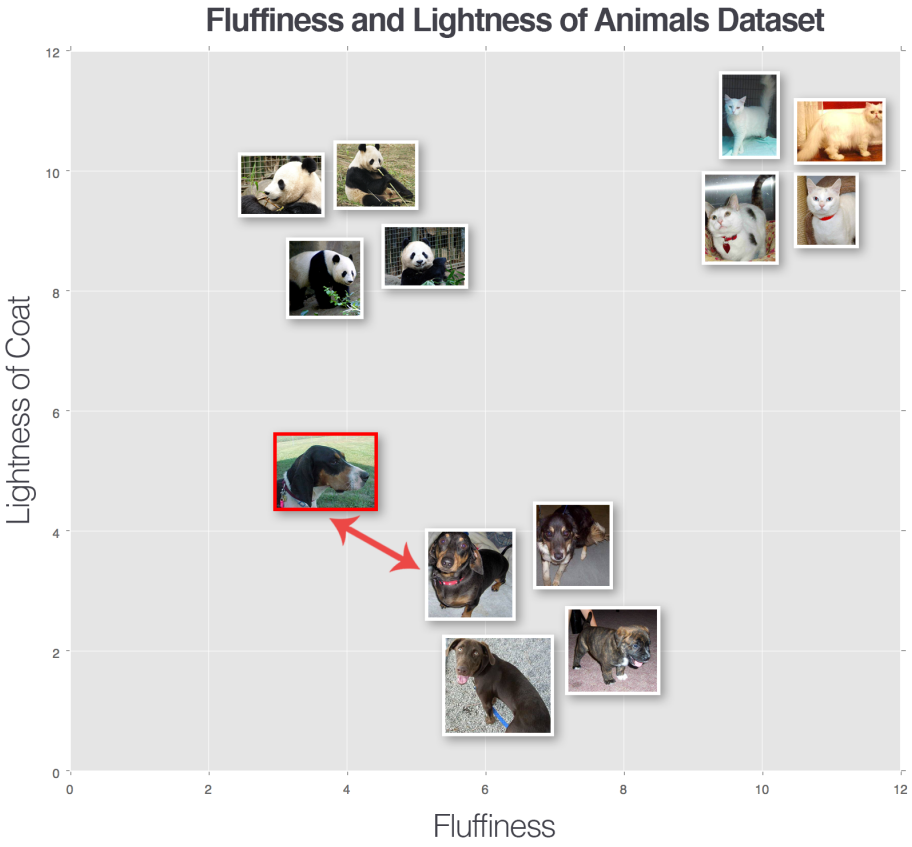


Figure 7.3: In this example we have inserted an unknown image (highlighted in red) into the dataset and then used the distance between the unknown animal and dataset of animals to make the classification.

Let's try another "unknown animal", this time using $k = 3$ (Figure 7.4). We have found *two cats* and *one panda* in the *top three results*. Since the *cat* category has the largest number of votes, we'll classify our input image as *cat*.

We can keep performing this process for varying values of k, but no matter how large or small k becomes, the principle remains the same – the category with the largest number of votes in the k closest training points wins and is used as the label for the input data point.

R In the event if a tie, the k-NN algorithm chooses one of the tied class labels at random.

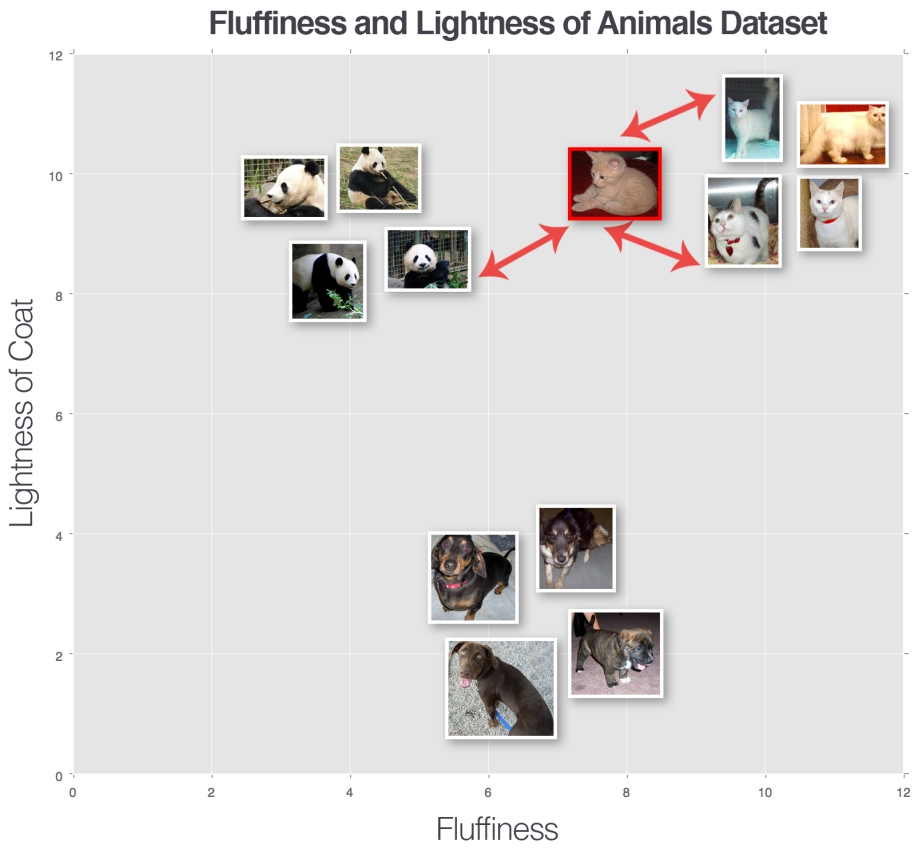


Figure 7.4: Classifying another animal, only this time we used $k = 3$ rather than just $k = 1$. Since there are two cat images closer to the input image than the single panda image, we'll label this input input image as *cat*.

7.2.2 k-NN Hyperparameters

There are two clear hyperparameters that we are concerned with when running the k-NN algorithm. The first is obvious: the value of k. What is the optimal value of k? If it's too small (such as $k = 1$), then we gain efficiency but become susceptible to noise and outlier data points. However, if k is too large, then we are at risk of *over-smoothing* our classification results and increasing bias.

The second parameter we should consider is the actual distance metric. Is the Euclidean distance the best choice? What about the Manhattan distance?

In the next section, we'll train our k-NN classifier on the Animals dataset and evaluate the model on our testing set. I would encourage you to play around with different values of `k` along with varying distance metrics, noting how performance changes.

For an exhaustive review of how to tune k-NN hyperparameters, please refer to Lesson 4.3 inside the PyImageSearch Gurus course [33].

7.2.3 Implementing k-NN

The goal of this section is to train a k-NN classifier on the *raw pixel intensities* of the Animals dataset and use it to classify unknown animal images. We'll be using our four step pipeline to train classifiers from Section 4.3.2:

- **Step #1 – Gather Our Dataset:** The Animals datasets consists of 3,000 images with 1,000 images per dog, cat, and panda class, respectively. Each image is represented in the RGB

color space. We will preprocess each image by resizing it to 32×32 pixels. Taking into account the three RGB channels, the resized image dimensions imply that each image in the dataset is represented by $32 \times 32 \times 3 = 3,072$ integers.

- **Step #2 – Split the Dataset:** For this simple example, we'll be using *two* splits of the data. One split for training, and the other for testing. We will leave out the validation set for hyperparameter tuning and leave this as an exercise to the reader.
- **Step #3 – Train the Classifier:** Our k-NN classifier will be trained on the raw pixel intensities of the images in the training set.
- **Step #4 – Evaluate:** Once our k-NN classifier is trained, we can evaluate performance on the test set.

Let's go ahead and get started. Open up a new file, name it `knn.py`, and insert the following code:

```
1  # import the necessary packages
2  from sklearn.neighbors import KNeighborsClassifier
3  from sklearn.preprocessing import LabelEncoder
4  from sklearn.model_selection import train_test_split
5  from sklearn.metrics import classification_report
6  from pyimagesearch.preprocessing import SimplePreprocessor
7  from pyimagesearch.datasets import SimpleDatasetLoader
8  from imutils import paths
9  import argparse
```

Lines 2-9 import our required Python packages. The most important ones to take note of are:

- **Line 2**: The `KNeighborsClassifier` is our implementation of the k-NN algorithm, provided by the scikit-learn library.
- **Line 3:** `LabelEncoder`, a helper utility to convert labels represented as strings to integers where there is one unique integer per class label (a common practice when applying machine learning).
- **Line 4:** We'll import the `train_test_split` function, which is a handy convenience function used to help us create our training and testing splits.
- **Line 5:** The `classification_report` function is another utility function that is used to help us evaluate the performance of our classifier and print a nicely formatted table of results to our console.

You can also see our implementations of the `SimplePreprocessor` and `SimpleDatasetLoader` imported on **Lines 6 and Line 7**, respectively.

Next, let's parse our command line arguments:

```
11 # construct the argument parse and parse the arguments
12 ap = argparse.ArgumentParser()
13 ap.add_argument("-d", "--dataset", required=True,
14     help="path to input dataset")
15 ap.add_argument("-k", "--neighbors", type=int, default=1,
16     help="# of nearest neighbors for classification")
17 ap.add_argument("-j", "--jobs", type=int, default=-1,
18     help="# of jobs for k-NN distance (-1 uses all available cores)")
19 args = vars(ap.parse_args())
```

Our script requires one command line argument, followed by two optional ones, each reviewed below:

- `--dataset`: The path to where our input image dataset resides on disk.
- `--neighbors`: Optional, the number of neighbors k to apply when using the k-NN algorithm.
- `--jobs`: Optional, the number of concurrent jobs to run when computing the distance between an input data point and the training set. A value of `-1` will use all available cores on the processor.

Now that our command line arguments are parsed, we can grab the file paths of the images in our dataset, followed by loading and preprocessing them (Step #1 in the classification pipeline):

```
21 # grab the list of images that we'll be describing
22 print("[INFO] loading images...")
23 imagePaths = list(paths.list_images(args["dataset"]))
24
25 # initialize the image preprocessor, load the dataset from disk,
26 # and reshape the data matrix
27 sp = SimplePreprocessor(32, 32)
28 sdl = SimpleDatasetLoader(preprocessors=[sp])
29 (data, labels) = sdl.load(imagePaths, verbose=500)
30 data = data.reshape((data.shape[0], 3072))
31
32 # show some information on memory consumption of the images
33 print("[INFO] features matrix: {:.1f}MB".format(
34     data.nbytes / (1024 * 1000.0)))
```

Line 23 grabs the file paths to all images in our dataset. We then initialize our `SimplePreprocessor` used to resize each image to 32×32 pixels on **Line 27**.

The `SimpleDatasetLoader` is initialized on **Line 28**, supplying our instantiated `SimplePreprocessor` as an argument (implying that `sp` will be applied to *every image* in the dataset).

A call to `.load` on **Line 29** loads our actual image dataset from disk. This method returns a 2-tuple of our `data` (each image resized to 32×32 pixels) along with the `labels` for each image.

After loading our images from disk, the `data` NumPy array has a `.shape` of `(3000, 32, 32, 3)`, indicating there are 3,000 images in the dataset, each 32×32 pixels with 3 channels.

However, in order to apply the k-NN algorithm, we need to "flatten" our images from a 3D representation to a single list of pixel intensities. We accomplish this, **Line 30** calls the `.reshape` method on the `data` NumPy array, flattening the $32 \times 32 \times 3$ images into an array with shape `(3000, 3072)`. The actual image data hasn't changed at all – the images are simply represented as a list of 3,000 entries, each of 3,072-dim ($32 \times 32 \times 3 = 3,072$).

To demonstrate how much memory it takes to store these 3,000 images in memory, **Lines 33 and 34** compute the number of bytes the array consumes and then converts the number to megabytes).

Next, let's building our training and testing splits (Step #2 in our pipeline):

```
36 # encode the labels as integers
37 le = LabelEncoder()
38 labels = le.fit_transform(labels)
39
40 # partition the data into training and testing splits using 75% of
41 # the data for training and the remaining 25% for testing
42 (trainX, testX, trainY, testY) = train_test_split(data, labels,
43     test_size=0.25, random_state=42)
```

Lines 37 and 38 convert our `labels` (represented as strings) to integers where we have *one unique integer* per class. This conversion allows us to map the *cat* class to the integer `0`, the *dog* class to integer `1`, and the *panda* class to integer `2`. Many machine learning algorithms assume that the class labels are encoded as integers, so it's important that we get in the habit of performing this step.

Computing our training and testing splits is handled by the `train_test_split` function on **Lines 42 and 43**. Here we partition our `data` and `labels` into two unique sets: 75% of the data for training and 25% for testing.

It is common to use the variable X to refer to a dataset that contains the data points we'll use for training and testing while y refers to the class labels (you'll learn more about this in Chapter 8 on *parameterized learning*). Therefore, we use the variables `trainX` and `testX` to refer to the *training and testing examples*, respectively. The variables `trainY` and `testY` are our *training and testing labels*. You will see these common notations throughout this book *and* in other machine learning books, courses, and tutorials that you may read.

Finally, we are able to create our k-NN classifier and evaluate it (Steps #3 and #4 in the image classification pipeline):

```
45 # train and evaluate a k-NN classifier on the raw pixel intensities
46 print("[INFO] evaluating k-NN classifier...")
47 model = KNeighborsClassifier(n_neighbors=args["neighbors"],
48      n_jobs=args["jobs"])
49 model.fit(trainX, trainY)
50 print(classification_report(testY, model.predict(testX),
51      target_names=le.classes_))
```

Lines 47 and 48 initialize the `KNeighborsClassifier` class. A call to the `.fit` method on **Line 49** "trains" the classifier, although there is no actual "learning" going on here – the k-NN model is simply storing the `trainX` and `trainY` data internally so it can create predictions on the testing set by computing the distance between the input data and the `trainX` data.

Lines 50 and 51 evaluate our classifier by using the `classification_report` function. Here we need to supply the `testY` class labels, the *predicted class labels* from our `model`, and optionally the names of the class labels (i.e., "dog", "cat", "panda").

7.2.4 k-NN Results

To run our k-NN classifier, execute the following command:

```
$ python knn.py --dataset ../datasets/animals
```

You should then see the following output similar to the following:

```
[INFO] loading images...
[INFO] processed 500/3000
[INFO] processed 1000/3000
[INFO] processed 1500/3000
[INFO] processed 2000/3000
[INFO] processed 2500/3000
[INFO] processed 3000/3000
[INFO] features matrix: 9.0MB
[INFO] evaluating k-NN classifier...
```

```
              precision    recall  f1-score   support

        cats       0.39      0.49      0.43       239
        dogs       0.36      0.47      0.41       249
       panda       0.79      0.36      0.50       262

avg / total       0.52      0.44      0.45       750
```

Notice how our feature matrix only consumes 9MB of memory for 3,000 images, each of size $32 \times 32 \times 3$ – this dataset can *easily* be stored in memory on modern machines without a problem.

Evaluating our classifier, we see that we obtained *52%* accuracy – this accuracy isn't bad for a classifier that doesn't do any true "learning" at all, given that the probability of randomly guessing the correct answer is $1/3$.

However, it is interesting to inspect the accuracy for each of the class labels. The "panda" class was correctly classified 79% of the time, likely due to the fact that pandas are largely black and white and thus these images lie closer together in our $3,072$-dim space.

Dogs and cats obtain substantially lower classification accuracy at 39% and 36%, respectively. These results can be attributed to the fact that dogs and cats can have very similar shades of fur coats and the color of their coats cannot be used to discriminate between them. Background noise (such as grass in a backyard, the color of a couch an animal is resting on, etc.) can also "confuse" the k-NN algorithm as its unable to learn any discriminating patterns between these species. This confusion is one of the primary drawbacks of the k-NN algorithm: *while it's simple, it is also unable to learn from the data*.

Our next chapter will discuss the concept of *parameterized learning* where we can actually learn patterns from the *images themselves* rather than assuming images with similar contents will group together in an n-dimensional space.

7.2.5 Pros and Cons of k-NN

One main advantage of the k-NN algorithm is that it's *extremely simple* to implement and understand. Furthermore, the classifier takes absolutely no time to train, since all we need to do is store our data points for the purpose of later computing distances to them and obtaining our final classification.

However, we pay for this simplicity at classification time. Classifying a new testing point requires a comparison to *every single data point* in our training data, which scales *O(N)*, making working with larger datasets computationally prohibitive.

We can combat this time cost by using **Approximate Nearest Neighbor** (ANN) algorithms (such as kd-trees [65], FLANN [66], random projections [67, 68, 69], etc.); however, using these algorithms require that we trade space/time complexity for the "correctness" of our nearest neighbor algorithm since we are performing an approximation. That said, in many cases, it is well worth the effort and small loss in accuracy to use the k-NN algorithm. This behavior is in contrast to most machine learning algorithms (and *all* neural networks), where we spend a large amount of time upfront training our model to obtain high accuracy, and, in turn, have *very fast* classifications at testing time.

Finally, the k-NN algorithm is more suited for low-dimensional feature spaces (which images are not). Distances in high-dimensional feature spaces are often unintuitive, which you can read more about in Pedro Domingo's excellent paper [70].

It's also important to note that the k-NN algorithm doesn't actually "learn" anything – the algorithm is not able to make itself smarter if it makes mistakes; it's simply relying on distances in an n-dimensional space to make the classification.

Given these cons, why bother even studying the k-NN algorithm? The reason is that the algorithm is *simple*. It's *easy to understand*. And most importantly, it gives us a *baseline* that we

can use to compare neural networks and Convolutional Neural Networks to as we progress through the rest of this book.

7.3 Summary

In this chapter, we learned how to build a simple image processor and load an image dataset into memory. We then discussed the *k-Nearest Neighbor classifier*, or *k-NN* for short.

The k-NN algorithm classifies unknown data points by comparing the unknown data point to each data point in the training set. The comparison is done using a distance function or similarity metric. Then, from the most k similar examples in the training set, we accumulate the number of "votes" for each label. The category with the highest number of votes "wins" and is chosen as the overall classification.

While simple and intuitive, the k-NN algorithm has a number of drawbacks. The first is that it doesn't actually "learn" anything – if the algorithm makes a mistake, it has no way to "correct" and "improve" itself for later classifications. Secondly, without specialized data structures, the k-NN algorithm scales linearly with the number of data points, making it not only practically challenging to use in high dimensions, but theoretically questionable in terms of its usage [70].

Now that we have obtained a baseline for image classification using the k-NN algorithm, we can move on the *parameterized learning*, the foundation on which all deep learning and neural networks are built on. Using parameterized learning, we can actually *learn* from our input data and discover underlying patterns. This process will enable us to build high accuracy image classifiers that blow the performance of k-NN out of the water.

8. Parameterized Learning

In our previous chapter we learned about the k-NN classifier – a machine learning model so simple that it doesn't do any actual "learning" at all. We simply have to store the training data inside the model, and then predictions are made at test time by comparing the testing data points to our training data.

We've already discussed many of the pros and cons of k-NN, but in the context of large-scale datasets and deep learning, the most prohibitive aspect of k-NN is *the data itself*. While training may be simple, testing is quite slow, with the bottleneck being the distance computation between vectors. Computing the distances between training and testing points scales linearly with the number of points in our dataset, making the method impractical when our datasets become quite large. And while we can apply Approximate Nearest Neighbor methods such as ANN [71], FLANN [66], or Annoy [72], to speed up the search, that still doesn't alleviate the problem that k-NN cannot function without maintaining a replica of data inside the instantiation (or at least have a pointer to training set on disk, etc.)

To see why storing an exact replica of the training data inside the model is an issue, consider training a k-NN model and then deploying it to a customer base of 100, 1,000, or even 1,000,000 users. If your training set is only a few megabytes, this may not be a problem – but if your training set is measured in *gigabytes* to *terabytes* (as is the case for many datasets that we apply deep learning to), you have a real problem on your hands.

Consider the training set of the ImageNet dataset [42] which includes over 1.2 million images. If we trained a k-NN model on this dataset and then tried to deploy it to a set of users, we would need these users to download the k-NN model which internally represents *replicas* of the 1.2 million images. Depending on how you compress and store the data, this model could measure in hundreds of gigabytes to terabytes in storage costs and network overhead. Not only is this a waste of resources, its also not optimal for constructing a machine learning model.

Instead, a more desirable approach would be to define a machine learning model that can *learn patterns* from our input data during training time (requiring us to spend more time on the training process), but have the benefit of being defined by a *small number of parameters* that can easily be used to represent the model, *regardless of training size*. This type of machine learning is called

parameterized learning, which is defined as:

> *"A learning model that summarizes data with a set of parameters of fixed size (independent of the number of training examples) is called a parametric model. No matter how much data you throw at the parametric model, it won't change its mind about how many parameters it needs."* – Russell and Norvig (2009) [73]

In this chapter, we'll review the concept of parameterized learning and discuss how to implement a simple linear classifier. As we'll see later in this book, parameterized learning is the cornerstone of modern day machine learning and deep learning algorithms.

Much of this chapter was inspired by Andrej Karpathy's excellent Linear Classification notes inside Stanford's cs231n class [74]. A big thank you to Karpathy and the rest of the cs231n teaching assistants for putting together such accessible notes.

8.1 An Introduction to Linear Classification

The first half of this chapter focuses on the fundamental theory and mathematics surrounding *linear classification* – and in general, *parameterized classification algorithms* that learn patterns from their training data. From there, I provide an actual linear classification implementation and example in Python so we can see how these types of algorithms work in code.

8.1.1 Four Components of Parameterized Learning

I've used the word "parameterized" a few times now, but what exactly does it mean?

Simply put: *parameterization* is the process of defining the necessary parameters of a given model. In the task of machine learning, parameterization involves defining a problem in terms of four key components: *data*, a *scoring function*, a *loss function*, and *weights and biases*. We'll review each of these below.

Data

This component is our *input data* that we are going to learn from. This data includes *both* the data points (i.e., raw pixel intensities from images, extracted features, etc.) and their associated class labels. Typically we denote our data in terms of a multi-dimensional **design matrix** [10].

Each row in the design matrix represents a data point while each column (which itself could be a multi-dimensional array) of the matrix corresponds to a different feature. For example, consider a dataset of 100 images in the RGB color space, each image sized 32×32 pixels. The design matrix for this dataset would be $X \subseteq R^{100 \times (32 \times 32 \times 3)}$ where X_i defines the i-th image in R. Using this notation, X_1 is the first image, X_2 the second image, and so on.

Along with the design matrix, we also define a vector y where y_i provides the class label for the i-th example in the dataset.

Scoring Function

The scoring function accepts our data as an input and maps the data to class labels. For instance, given our set of input images, the scoring function takes these data points, applies some function f (our scoring function), and then returns the predicted class labels, similar to the pseudocode below:

```
INPUT_IMAGES => F(INPUT_IMAGES) => OUTPUT_CLASS_LABELS
```

Loss Function

A loss function quantifies how well our *predicted class labels* agree with our *ground-truth labels*. The higher level of agreement between these two sets of labels, the *lower our loss* (and higher our classification accuracy, at least on the training set).

Our goal when training a machine learning model is to *minimize the loss function*, thereby increasing our classification accuracy.

Weights and Biases

The weight matrix, typically denoted as $\boldsymbol{W}$ and the bias vector $\boldsymbol{b}$ are called the **weights** or **parameters** of our classifier that we'll actually be optimizing. Based on the output of our scoring function and loss function, we'll be tweaking and fiddling with the values of the weights and biases to increase classification accuracy.

Depending on your model type, there may exist many more parameters, but at the most basic level, these are the four building blocks of parameterized learning that you'll commonly encounter. Once we've defined these four key components, we can then apply optimization methods that allow us to find a set of parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ that minimize our loss function with respect to our scoring function (while increasing classification accuracy on our data).

Next, let's look at how these components can work together to build a linear classifier, transforming the input data into actual predictions.

8.1.2 Linear Classification: From Images to Labels

In this section, we are going to look at a more mathematical motivation of the parameterized model approach to machine learning.

To start, we need our ***data***. Let's assume that our training dataset is denoted as x_i where each image has an associated class label y_i. We'll assume that $i = 1, \ldots, N$ and $y_i = 1, \ldots, K$, implying that we have N data points of dimensionality D, separated into K unique categories.

To make this idea more concrete, consider our "Animals" dataset from Chapter 7. In this dataset, we have $N = 3{,}000$ total images. Each image is 32×32 pixels, represented in the RGB color space (i.e,. three channels per image). We can represent each image as $D = 32 \times 32 \times 3 = 3{,}072$ distinct values. Finally, we know there are a total of $K = 3$ class labels: one for the dog, cat, and panda classes, respectively.

Given these variables, we must now define a scoring function f that maps the images to the class label scores. One method to accomplish this scoring is via a simple linear mapping:

$$f(x_i, W, b) = Wx_i + b \tag{8.1}$$

Let's assume that each x_i is represented as a single column vector with shape $[D \times 1]$ (in this example we would flatten the $32 \times 32 \times 3$ image into a list of 3,072 integers). Our weight matrix $\boldsymbol{W}$ would then have a shape of $[K \times D]$ (the number of class labels by the dimensionality of the input images). Finally $\boldsymbol{b}$, the **bias vector** would be of size $[K \times 1]$. The bias vector allows us to shift and translate our scoring function in one direction or another without actually influencing our weight matrix $\boldsymbol{W}$. The bias parameter is often critical for successful learning.

Going back to the Animals dataset example, each x_i is represented by a list of 3,072 pixel values, so x_i, therefore, has the shape $[3{,}072 \times 1]$. The weight matrix $\boldsymbol{W}$ will have a shape of $[3 \times 3{,}072]$ and finally the bias vector $\boldsymbol{b}$ will be of size $[3 \times 1]$.

Figure 8.1 follows an illustration of the linear classification scoring function f. On the *left*, we have our original input image, represented as a $32 \times 32 \times 3$ image. We then *flatten* this image into a list of 3,072 pixel intensities by taking the 3D array and reshaping it into a 1D list.

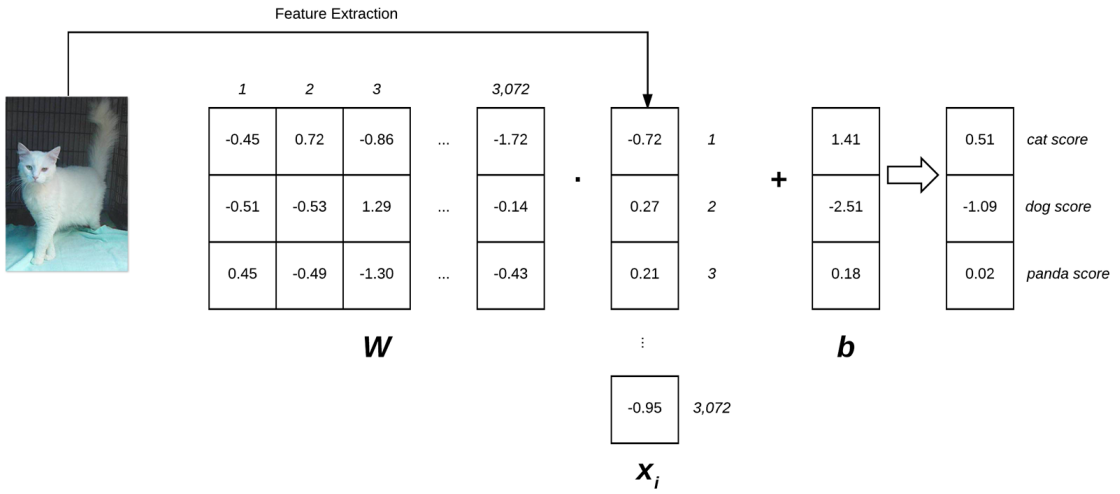


Figure 8.1: Illustrating the dot product of the weight matrix $\boldsymbol{W}$ and feature vector $\boldsymbol{x}$, followed by the addition of the bias term. Figure inspired by Karpathy's example in Stanford University's cs231n course [57].

Our weight matrix $\boldsymbol{W}$ contains three rows (one for each class label) and 3,072 columns (one for each of the pixels in the image). After taking the dot product between $\boldsymbol{W}$ and x_i, we add in the bias vector $\boldsymbol{b}$ – the result is our actual **scoring function**. Our scoring function yields three values on the *right*: the scores associated with the dog, cat, and panda labels, respectively.

Readers who are unfamiliar with taking dot products should read this quick and concise tutorial: http://pyimg.co/fgcvp. For readers interested in studying linear algebra in depth, I highly recommend working through *Coding the Matrix Linear Algebra through Applications to Computer Science* by Philip N. Klein [75].

Looking at the above figure and equation, you can convince yourself that the input x_i and y_i are *fixed* and *not something we can modify*. Sure, we can obtain different x_is by applying various transformations to the input image – but once we pass the image into the scoring function, **these values do not change**. In fact, the only parameters that we have any control over (in terms of parameterized learning) are our weight matrix $\boldsymbol{W}$ and our bias vector $\boldsymbol{b}$. Therefore, our goal is to utilize both our scoring function and loss function to *optimize* (i.e., modify in a systematic way) the weight and bias vectors such that our classification accuracy *increases*.

Exactly *how* we optimize the weight matrix depends on our loss function, but typically involves some form of gradient descent. We'll be reviewing loss functions later in this chapter. Optimization methods such as gradient descent (and its variants) will be discussed in Chapter 9. However, for the time being, simply understand that given a scoring function, we will also define a loss function that tells us how "good" our predictions are on the input data.

8.1.3 Advantages of Parameterized Learning and Linear Classification

There are two primary advantages to utilizing parameterized learning:

1. **Once we are done training our model, we can discard the input data and keep only the weight matrix $\boldsymbol{W}$ and the bias vector $\boldsymbol{b}$.** This *substantially* reduces the size of our model since we need to store two sets of vectors (versus the *entire* training set).
2. **Classifying new test data is *fast*.** In order to perform a classification, all we need to do is take the dot product of $\boldsymbol{W}$ and x_i, follow by adding in the bias $\boldsymbol{b}$ (i.e., apply our scoring

function). Doing it this way is *significantly faster* than needing to compare each testing point to *every* training example, as in the k-NN algorithm.

8.1.4 A Simple Linear Classifier With Python

Now that we've reviewed the concept of parameterized learning and linear classification, let's implement a *very simple* linear classifier using Python.

The purpose of this example is *not* to demonstrate how we train a model from start to finish (we'll be covering that in a later chapter as we still have some ground to cover before we're ready to train a model from scratch), but to simply show how we would initialize a weight matrix $\boldsymbol{W}$, bias vector $\boldsymbol{b}$, and then use these parameters to classify an image via a simple dot product.

Let's go ahead and get this example started. Our goal here is to write a Python script that will correctly classify Figure 8.2 as *"dog"*.

Figure 8.2: Our example input image that we are going to classifier with a simple linear classifier.

To see how we can accomplish this classification, open a new file, name it `linear_example.py`, and insert the following code:

```
1  # import the necessary packages
2  import numpy as np
3  import cv2
4
5  # initialize the class labels and set the seed of the pseudorandom
6  # number generator so we can reproduce our results
7  labels = ["dog", "cat", "panda"]
8  np.random.seed(1)
```

Lines 2 and 3 import our required Python packages. We'll use NumPy for our numerical processing and OpenCV to load our example image from disk.

Line 7 initializes the list of target class labels for the "Animals" dataset while **Line 8** sets the pseudorandom number generator for NumPy, ensuring that we can reproduce the results of this experiment.

Next, let's initialize our weight matrix and bias vector:

```
10 # randomly initialize our weight matrix and bias vector -- in a
11 # *real* training and classification task, these parameters would
```

```
12 # be *learned* by our model, but for the sake of this example,
13 # let's use random values
14 W = np.random.randn(3, 3072)
15 b = np.random.randn(3)
```

Line 14 initializes the weight matrix `W` with random values from a uniform distribution, sampled over the range $[0, 1]$. This weight matrix has `3` rows (one for each of the class labels) and `3072` columns (one for each of the pixels in our $32 \times 32 \times 3$ image).

We then initialize the bias vector on **Line 15** – this vector is also randomly filled with values uniformly sampled over the distribution $[0, 1]$. Our bias vector has `3` rows (corresponding to the number of class labels) along with one column.

If we were training this linear classifier *from scratch* we would need to *learn* the values of `W` and `b` through an optimization process. However, since we have not reached the optimization stage of training a model, I have initialized the pseudorandom number generator with a value `1` to ensure the random values give us the "correct" classification (I tested random initialization values ahead of time to determine which value gives us the correct classification). For the time being, simply treat the weight matrix `W` and the bias vector `b` as "black box arrays" that are optimized in a magical way – we'll pull back the curtain and reveal how these parameters are learned in the next chapter.

Now that our weight matrix and bias vector are initialized, let's load our example image from disk:

```
17 # load our example image, resize it, and then flatten it into our
18 # "feature vector" representation
19 orig = cv2.imread("beagle.png")
20 image = cv2.resize(orig, (32, 32)).flatten()
```

Line 19 loads our image from disk via `cv2.imread`. We then resize the image to 32×32 pixels (ignoring the aspect ratio) on **Line 20** – our image is now represented as a `(32, 32, 3)` NumPy array, which we flatten into 3,072-dim vector.

The next step is to compute the output class label scores by applying our scoring function:

```
22 # compute the output scores by taking the dot product between the
23 # weight matrix and image pixels, followed by adding in the bias
24 scores = W.dot(image) + b
```

Line 24 is the scoring function itself – it's simply the dot product between the weight matrix `W` and the input image pixel intensities, followed by adding in the bias `b`.

Finally, our last code block handles writing the scoring function values for each of the class labels to our terminal, then displaying the result to our screen:

```
26 # loop over the scores + labels and display them
27 for (label, score) in zip(labels, scores):
28     print("[INFO] {}: {:.2f}".format(label, score))
29
30 # draw the label with the highest score on the image as our
31 # prediction
32 cv2.putText(orig, "Label: {}".format(labels[np.argmax(scores)]),
33     (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)
34
```

```
35 # display our input image
36 cv2.imshow("Image", orig)
37 cv2.waitKey(0)
```

To execute our example, just issue the following command:

```
$ python linear_example.py
[INFO] dog: 7963.93
[INFO] cat: -2930.99
[INFO] panda: 3362.47
```

Notice how the *dog* class has the *largest scoring function value*, which implies that the "dog" class would be chosen as the prediction by our classifier. In fact, we can see the text `dog` correctly drawn on our input image (Figure 8.2) in Figure 8.3.

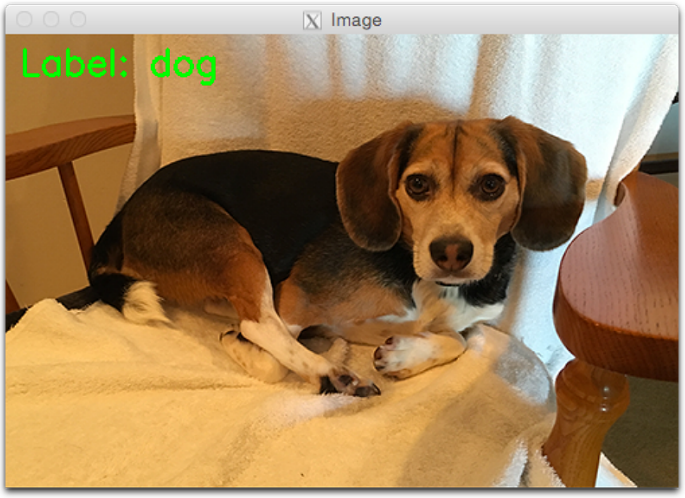


Figure 8.3: In this example, our linear classifier was correctly able to label the input image as *dog*; however, keep in mind that this is a worked example. Later in this book you'll learn how to *train* our weights and biases to automatically make these predictions.

Again, keep in mind that this was a *worked example*. I ***purposely*** set the random state of our Python script to generate `W` and `b` values that would lead to the correct classification (you can change the pseudorandom seed value on **Line 8** to see for yourself how different random initializations will produce different output predictions).

In practice, you would *never* initialize your `W` and `b` values and *assume* they would give you the correct classification without some sort of learning process. Instead, when training our own machine learning models from scratch we would need to *optimize* and *learn* `W` and `b` via an optimization algorithm, such as gradient descent.

We'll cover optimization and gradient descent in the next chapter, but in the meantime, simply take the time to ensure you understand **Line 24** and how a linear classifier makes a classification by taking the dot product between a weight matrix and an input data point, followed by adding in the bias. Our *entire model* can therefore be defined via two values: the weight matrix and the bias vector. This representation is not only *compact* but also quite *powerful* when we train machine learning models from scratch.

8.2 The Role of Loss Functions

In our last section we discussed the concept of parameterized learning. This type of learning allows us to take sets of input data and class labels, and actually learn a function that *maps* the input to the output predictions by defining a set of parameters and optimizing over them.

But in order to actually "learn" the mapping from the input data to class labels via our scoring function, we need to discuss two important concepts:

1. Loss functions
2. Optimization methods

The rest of this chapter is dedicated to common loss functions you'll encounter when building neural networks and deep learning networks. Chapter 9 of the *Starter Bundle* is dedicated entirely to basic optimization methods while Chapter 7 of the *Practitioner Bundle* discusses more advanced optimization methods.

Again, this chapter is meant to be a *brief review* of loss functions and their role in parameterized learning. A thorough discussion of loss functions is outside the scope of this book and I would highly recommend Andrew Ng's Coursera course [76], Witten et al. [77], Harrington [78], and Marsland [79] if you would like to complement this chapter with more mathematically rigorous derivations.

8.2.1 What Are Loss Functions?

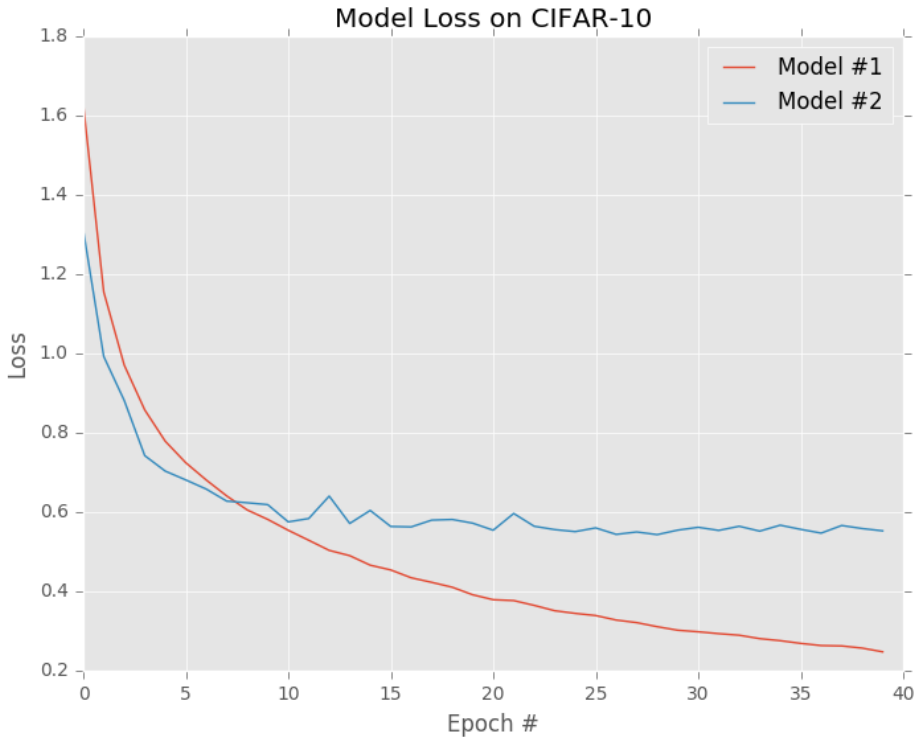


Figure 8.4: The training losses for two separate models trained on the CIFAR-10 dataset are plotted over time. Our loss function quantifies how "good" or "bad" of a job a given model is doing at classifying data points from the dataset. Model #1 achieves considerably lower loss than Model #2.

At the most basic level, a loss function quantifies how "good" or "bad" a given predictor is at classifying the input data points in a dataset. A visualization of loss functions plotted over time

for two separate models trained on the CIFAR-10 dataset is shown in Figure 8.4. The smaller the loss, the better a job the classifier is at modeling the relationship between the input data and output class labels (although there is a point where we can *overfit* our model – by modeling the training data *too closely*, our model loses the ability to generalize, a phenomenon we'll discuss in detail in Chapter 17). Conversely, the larger our loss, the *more work needs to be done* to increase classification accuracy.

To improve our classification accuracy, we need to tune the parameters of our weight matrix $\boldsymbol{W}$ or bias vector $\boldsymbol{b}$. Exactly *how* we go about updating these parameters is an *optimization problem*, which we'll be covering in the next chapter. For the time being, simply understand that a *loss function* can be used to quantify how well our *scoring function* is doing at classifying input data points.

Ideally, our loss should decrease over time as we tune our model parameters. As Figure 8.4 demonstrates, Model #1's loss starts slightly higher than Model #2, but then decreases rapidly and continues to stay low when trained on the CIFAR-10 dataset. Conversely, the loss for Model #2 decreases initially but quickly stagnates. In this specific example, Model #1 is achieving lower overall loss and is likely a more desirable model to be used on classifying other images from the CIFAR-10 dataset. I say "likely" because there is a chance that Model #1 has overfit to the training data. We'll cover this concept of overfitting and how to spot it in Chapter 17.

8.2.2 Multi-class SVM Loss

Multi-class SVM Loss (as the name suggests) is inspired by (Linear) Support Vector Machines (SVMs) [43] which uses a scoring function f to map our data points to numerical scores for each class labels. This function f is a simple learning mapping:

$$f(x_i, W, b) = Wx_i + b \tag{8.2}$$

Now that we have our scoring function, we need to determine how "good" or "bad" this function is (given the weight matrix $\boldsymbol{W}$ and bias vector $\boldsymbol{b}$) at making predictions. To make this determination, we need a *loss function*.

Recall that when creating a machine learning model we have a *design matrix* $\boldsymbol{X}$, where each row in $\boldsymbol{X}$ contains a data point we wish to classify. In the context of image classification, each row in $\boldsymbol{X}$ is an image and we seek to correctly label this image. We can access the i-th image inside $\boldsymbol{X}$ via the syntax x_i.

Similarly, we also have a vector $\boldsymbol{y}$ which contains our class labels for each $\boldsymbol{X}$. These $\boldsymbol{y}$ values are our *ground-truth labels* and what we hope our scoring function will correctly predict. Just like we can access a given image as x_i, we can access the associated class label via y_i.

As a matter of simplicity, let's abbreviate our scoring function as s:

$$s = f(x_i, W) \tag{8.3}$$

Which implies that we can obtain the predicted score of the j-th class via the i-th data point:

$$s_j = f(x_i, W)_j \tag{8.4}$$

Using this syntax, we can put it all together, obtaining the ***hinge loss function***:

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1) \tag{8.5}$$

Nearly all loss functions include a regularization term. I am skipping this idea for now as we'll review regularization in Chapter 9 once we better understand loss functions.

Looking at the hinge loss equation above, you might be confused at what it's actually doing. Essentially, the hinge loss function is summing across all *incorrect classes* ($i \neq j$) and comparing the output of our scoring function s returned for the j-th class label (the incorrect class) and the y_i-th class (the correct class). We apply the *max* operation to clamp values at zero, which is important to ensure we do not sum negative values.

A given x_i is classified correctly when the loss $L_i = 0$ (I'll provide a numerical example in the following section). To derive the loss across our *entire training set*, we simply take the mean over each individual L_i:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i \tag{8.6}$$

Another related loss function you may encounter is the ***squared hinge loss***:

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)^2 \tag{8.7}$$

The squared term penalizes our loss more heavily by squaring the output, which leads to quadratic growth in loss in a prediction that is incorrect (versus a linear growth).

As for which loss function you should use, that is entirely dependent on your dataset. It's typical to see the standard hinge loss function used more, but on some datasets the squared variation may obtain better accuracy. Overall, this is a *hyperparameter* that you should consider tuning.

A Multi-class SVM Loss Example

Now that we've taken a look at the mathematics behind hinge loss, let's examine a worked example. We'll again be using the "Animals" dataset which aims to classify a given image as containing a *cat*, *dog*, or *panda*. To start, take a look at Figure 8.5 where I have included three training examples from the three classes of the "Animals" dataset.

Given some arbitrary weight matrix $\boldsymbol{W}$ and bias vector $\boldsymbol{b}$, the output scores of $f(x, W) = Wx + b$ are displayed in the body of the matrix. The *larger* the scores are, the *more confident* our scoring function is regarding the prediction.

Let's start by computing the loss L_i for the "dog" class:

```
>>> max(0, 1.33 - 4.26 + 1) + max(0, -1.01 - 4.26 + 1)
0
```

Notice how our equation here includes two terms – the difference between the predicted dog score and *both* the cat and panda score. Also observe how the loss for "dog" is *zero* – this implies that the dog was correctly predicted. A quick investigation of Image #1 from Figure 8.5 above demonstrates this result to be true: the "dog" score is greater than both the "cat" and "panda" scores.

Similarly, we can compute the hinge loss for Image #2, this one containing a cat:

```
>>> max(0, 3.76 - (-1.20) + 1) + max(0, -3.81 - (-1.20) + 1)
5.96
```



	Image #1	Image #2	Image #3
Dog	4.26	3.76	-2.37
Cat	1.33	-1.20	1.03
Panda	-1.01	-3.81	-2.27

Figure 8.5: At the top of the figure we have three input images: one for each of the dog, cat, and panda class, respectively. The body of the table contains the scoring function outputs for each of the classes. We will use the scoring function to derive the total loss for each input image.

In this case, our loss function is greater than zero, indicating that our prediction is *incorrect*. Looking at our scoring function, we see that our model predicts *dog* as the proposed label with a score of 3.76 (as this is the label with the highest score). We know that this label is incorrect – and in Chapter 9 we'll learn how to automatically tune our weights to correct these predictions.

Finally, let's compute the hinge loss for the panda example:

```
>>> max(0, -2.37 - (-2.27) + 1) + max(0, 1.03 - (-2.27) + 1)
5.199999999999999
```

Again, our loss is non-zero, so we know we have an incorrect prediction. Looking at our scoring function, our model has incorrectly labeled this image as "cat" when it should be "panda".

We can then obtain the *total loss* over the three examples by taking the average:

```
>>> (0.0 + 5.96 + 5.2) / 3.0
3.72
```

Therefore, given our three training examples our overall hinge loss is 3.72 for the parameters $\boldsymbol{W}$ and $\boldsymbol{b}$.

Also take note that our loss was zero for only *one* of the three input images, implying that *two* of our predictions were incorrect. In our next chapter we'll learn how to optimize $\boldsymbol{W}$ and $\boldsymbol{b}$ to make better predictions by using the loss function to help drive and steer us in the right direction.

8.2.3 Cross-entropy Loss and Softmax Classifiers

While hinge loss is quite popular, you're *much* more likely to run into cross-entropy loss and Softmax classifiers in the context of deep learning and convolutional neural networks.

Why is this? Simply put:

Softmax classifiers give you *probabilities* for each class label while hinge loss gives you the *margin*.

It's much easier for us as humans to interpret *probabilities* rather than margin scores. Furthermore, for datasets such as ImageNet, we often look at the rank-5 accuracy of Convolutional Neural Networks (where we check to see if the ground-truth label is in the top-5 predicted labels returned by a network for a given input image). Seeing if (1) the true class label exists in the top-5 predictions and (2) the *probability* associated with each label is a nice property.

Understanding Cross-entropy Loss

The Softmax classifier is a generalization of the binary form of Logistic Regression. Just like in hinge loss or squared hinge loss, our mapping function f is defined such that it takes an input set of data x_i and maps them to output class labels via dot product of the data x_i and weight matrix $\boldsymbol{W}$ (omitting the bias term for brevity):

$$f(x_i, W) = Wx_i \tag{8.8}$$

However, unlike hinge loss, we can interpret these scores as *unnormalized log probabilities* for each class label, which amounts to swapping out the hinge loss function with cross-entropy loss:

$$L_i = -log(e^{s_{y_i}} / \sum_j e^{s_j}) \tag{8.9}$$

So, how did I arrive here? Let's break the function apart and take a look. To start, our loss function should minimize the negative log likelihood of the correct class:

$$L_i = -logP(Y = y_i | X = x_i) \tag{8.10}$$

The probability statement can be interpreted as:

$$P(Y = k | X = x_i) = e^{s_{y_i}} / \sum_j e^{s_j} \tag{8.11}$$

Where we use our standard scoring function form:

$$s = f(x_i, W) \tag{8.12}$$

As a whole, this yields our final loss function for a *single* data point, just like above:

$$L_i = -log(e^{s_{y_i}} / \sum_j e^{s_j}) \tag{8.13}$$

Take note that your logarithm here is actually base e (natural logarithm) since we are taking the inverse of the exponentiation over e earlier. The actual exponentiation and normalization via the sum of exponents is our *Softmax function*. The negative log yields our actual *cross-entropy loss*.

Just as in hinge loss and squared hinge loss, computing the cross-entropy loss over an entire dataset is done by taking the average:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i \tag{8.14}$$

Again, I'm purposely omitting the regularization term from our loss function. We'll return to regularization, explain what it is, how to use it, and why it's critical to neural networks and deep learning in Chapter 9. If the equations above seem scary, don't worry – we're about to work through numerical examples in the next section to ensure you understand how cross-entropy loss works.

A Worked Softmax Example

	Scoring Function
Dog	-3.44
Cat	1.16
Panda	3.91

Input Image

	Scoring Function	Unnormalized Probabilities
Dog	-3.44	0.03
Cat	1.16	3.19
Panda	3.91	49.90

	Scoring Function	Unnormalized Probabilities	Normalized Probabilities
Dog	-3.44	0.0321	0.0006
Cat	1.16	3.1899	0.0601
Panda	3.91	49.8990	0.9393

	Scoring Function	Unnormalized Probabilities	Normalized Probabilities	Negative Log Loss
Dog	-3.44	0.0321	0.0006	
Cat	1.16	3.1899	0.0601	
Panda	3.91	49.8990	0.9393	**0.0626**

Figure 8.6: **First Table:** To compute our cross-entropy loss, let's start with the output of the scoring function. **Second Table:** Exponentiating the output values from the scoring function gives us our unnormalized probabilities. **Third Table:** To obtain the actual probabilities, we divide each individual unnormalized probabilities by the sum of all unnormalized probabilities. **Fourth Table:** Taking the negative natural logarithm of the probability for the correct ground-truth yields the final loss for the data point.

To demonstrate cross-entropy loss in action, consider Figure 8.6. Our goal is to classify whether the image above contains a *dog*, *cat*, or *panda*. Clearly, we can see that the image is a "panda" – *but what does our Softmax classifier think?* To find out, we'll need to work through each of the four tables in the figure.

The **first table** includes the output of our scoring function f for each of the three classes, respectively. These values are our *unnormalized log probabilities* for the three classes. Let's exponentiate the output of the scoring function (e^s, where s is our score function value), yielding our *unnormalized probabilities* (**second table**).

The next step is to take the denominator, sum the exponents, and divide by the sum, thereby yielding the *actual probabilities associated with each class label* (**third table**). Notice how the probabilities sum to one. Finally, we can take the negative natural logarithm, $-ln(p)$, where p is the normalized probability, yielding our final loss (the **fourth and final table**).

In this case, our Softmax classifier would correctly report the image as *panda* with 93.93% confidence. We can then repeat this process for all images in our training set, take the average, and obtain the overall cross-entropy loss for the training set. This process allows us to quantify how good or bad a set of parameters are performing on our training set.

I used a random number generator to obtain the score function values for this particular example. These values are simply used to demonstrate how the calculations of the Softmax classifier/cross-entropy loss function are performed. In reality, these values would *not* be randomly generated – they would instead be the output of your scoring function f based on your parameters $\boldsymbol{W}$ *and* $\boldsymbol{b}$. We'll see how all the components of parameterized learning fit together in our next chapter, but for the time being, we are working with example numbers to demonstrate how loss functions work.

8.3 Summary

In this chapter, we reviewed four components of parameterized learning:

1. Data
2. Scoring function
3. Loss function
4. Weights and biases

In the context of image classification, our input data is our dataset of images. The scoring function produces *predictions* for a given input image. The loss function then *quantifies* how good or bad a set of predictions are over the dataset. Finally, the weight matrix and bias vectors are what enable us to actually "learn" from the input data – these parameters will be tweaked and tuned via optimization methods in an attempt to obtain higher classification accuracy.

We then reviewed two popular loss functions: *hinge loss* and *cross-entropy loss*. While hinge loss is used in many machine learning applications (such as SVMs), I can almost guarantee with absolutely certainty that you'll see cross-entropy loss with more frequency primarily due to the fact that Softmax classifiers output *probabilities* rather than *margins*. Probabilities are much easier for us as humans to interpret, so this fact is a particularly nice quality of cross-entropy loss and Softmax classifiers. For more information on loss hinge loss and cross-entropy loss, please refer to Stanford University's cs231n course [57, 74].

In our next chapter we'll review optimization methods that are used to tune our weight matrix and bias vector. Optimization methods allow our algorithms to actually *learn* from our input data by updating the weight matrix and bias vector based on the output of our scoring and loss functions. Using these techniques we can take *incremental steps* towards parameter values that obtain lower loss and higher accuracy. Optimization methods are the cornerstone of modern day neural networks and deep learning, and without them, we would be unable to learn patterns from our input data, so be sure to pay attention to the upcoming chapter.

9. Optimization Methods and Regularization

"Nearly all of deep learning is powered by one very important algorithm: Stochastic Gradient Descent (SGD)" – Goodfellow et al. [10]

At this point we have a strong understanding of the concept of parameterized learning. Over the past few chapters, we have discussed the concept of *parameterized learning* and how this type of learning enables us to define a *scoring function* that maps our input data to output class labels.

This scoring function is defined in terms of two important *parameters*; specifically, our weight matrix $\boldsymbol{W}$ and our bias vector $\boldsymbol{b}$. Our scoring function accepts these parameters as inputs and returns a prediction for each input data point x_i.

We have also discussed two common loss functions: Multi-class SVM loss and cross-entropy loss. Loss functions, at the most basic level, are used to quantify how "good" or "bad" a given predictor (i.e., a set of parameters) is at classifying the input data points in our data.

Given these building blocks, we can now move on to the *most important aspect* of machine learning, neural networks, and deep learning – ***optimization***. Optimization algorithms are the engines that power neural networks and enable them to learn patterns from data. Throughout this discussion, we've learned that obtaining a high accuracy classifier is *dependent* on finding a set of weights $\boldsymbol{W}$ and $\boldsymbol{b}$ such that our data points are correctly classified.

But how do we go about *finding* and *obtaining* a weight matrix $\boldsymbol{W}$ and bias vector $\boldsymbol{b}$ that obtains high classification accuracy? Do we randomly initialize them, evaluate, and repeat over and over again, *hoping* that at *some point* we land on a set of parameters that obtains reasonable classification? We could – but given that modern deep learning networks have parameters that number in the tens of millions, it may take us a long time to blindly stumble upon a reasonable set of parameters.

Instead of relying on pure randomness, we need to define an *optimization algorithm* that allows us to *literally improve* $\boldsymbol{W}$ **and** $\boldsymbol{b}$. In this chapter, we'll be looking at the most common algorithm used to train neural networks and deep learning models – *gradient descent*. Gradient descent has many variants (which we'll also touch on), but, in each case, the idea is the same: iteratively evaluate your parameters, compute your loss, then take a small step in the direction that will minimize your loss.

9.1 Gradient Descent

The gradient descent algorithm has two primary flavors:

1. The standard "vanilla" implementation.
2. The optimized "stochastic" version that is more commonly used.

In this section we'll be reviewing the basic vanilla implementation to form a baseline for our understanding. After we understand the basics of gradient descent, we'll move on to the stochastic version. We'll then review some of the "bells and whistles" that we can add on to gradient descent, including momentum, and Nesterov acceleration.

9.1.1 The Loss Landscape and Optimization Surface

The gradient descent method is an *iterative optimization algorithm* that operates over a **loss landscape** (also called an *optimization surface*). The canonical gradient descent example is to visualize our weights along the x-axis and then the loss for a given set of weights along the y-axis (Figure 9.1, *left*):

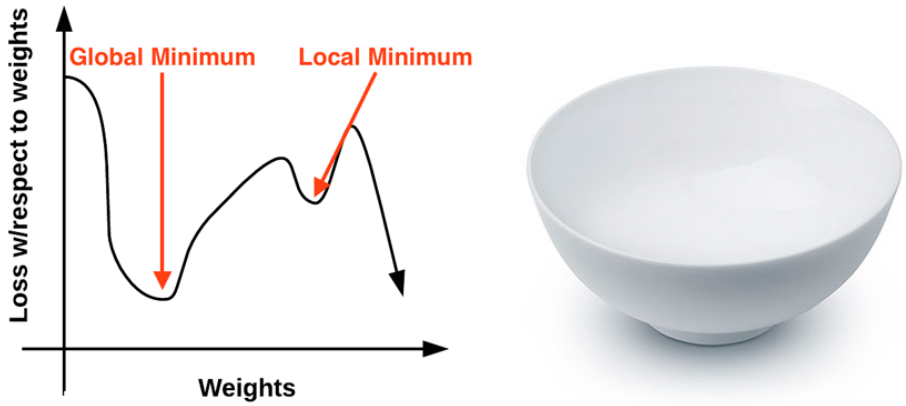


Figure 9.1: **Left:** The "naive loss" visualized as a 2D plot. **Right:** A more realistic loss landscape can be visualized as a bowl that exists in multiple dimensions. Our goal is to apply gradient descent to navigate to the bottom of this bowl (where there is low loss).

As we can see, our loss landscape has many peaks and valleys based on which values our parameters take on. Each peak is a *local maximum* that represents very high regions of loss – the local maximum with the largest loss across the entire loss landscape is the *global maximum*. Similarly, we also have *local minimum* which represents many small regions of loss.

The local minimum with the smallest loss across the loss landscape is our *global minimum*. In an ideal world, we would like to find this global minimum, ensuring our parameters take on the most optimal possible values.

So that raises the question: *"If we want to reach a global minimum, why not just directly jump to it? It's clearly visible on the plot?"*

Therein lies the problem – the loss landscape is invisible to us. We don't *actually* know what it looks like. If we're an optimization algorithm, we would be *blindly* placed somewhere on the plot, having no idea what the landscape in front of us looks like, and we would have to navigate our way to a loss minimum without accidentally climbing to the top of a local maximum.

Personally, I've never liked this visualization of the loss landscape – it's too simple, and it often leads readers to think that gradient descent (and its variants) will eventually find either a local or global minimum. *This statement isn't true, especially for complex problems* – and I'll explain why

later in this chapter. Instead, let's look at a different visualization of the loss landscape that I believe does a better job depicting the problem. Here we have a bowl, similar to the one you may eat cereal or soup out of (Figure 9.1, *right*).

The surface of our bowl is the loss landscape, which is a *plot* of the loss function. The difference between our loss landscape and your cereal bowl is that your cereal bowl only exists in three dimensions, while your loss landscape exists in *many dimension*, perhaps tens, hundreds, or thousands of dimensions.

Each position along the surface of the bowl corresponds to a *particular loss value* given a set of parameters $\boldsymbol{W}$ (weight matrix) and $\boldsymbol{b}$ (bias vector). Our goal is to try different values of $\boldsymbol{W}$ and $\boldsymbol{b}$, evaluate their loss, and then take a step towards more optimal values that (ideally) have lower loss.

9.1.2 The "Gradient" in Gradient Descent

To make our explanation of gradient descent a little more intuitive, let's pretend that we have a robot – let's name him Chad (Figure 9.2, *left*). When performing gradient descent, we randomly drop Chad somewhere on our loss landscape (Figure 9.2, *right*).

Figure 9.2: **Left:** Our robot, Chad. **Right:** It's Chad's job to navigate our loss landscape and descend to the bottom of the basin. Unfortunately, the only sensor Chad can use to control his navigation is a special function, called a *loss function*, L. This function must guide him to an area of lower loss.

It's now Chad's job to navigate to the bottom of the basin (where there is minimum loss). Seems easy enough right? All Chad has to do is orient himself such that he's facing "downhill" and ride the slope until he reaches the bottom of the bowl.

But here's the problem: Chad isn't a very smart robot. Chad has only one sensor – this sensor allows him to take his parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ and then compute a loss function L. Therefore, Chad is able to compute his relative position on the loss landscape, but he has *absolutely no idea* in which direction he should take a step to move himself closer to the bottom of the basin.

What is Chad to do? ***The answer is to apply gradient descent.*** All Chad needs to do is follow the slope of the gradient $\boldsymbol{W}$. We can compute the gradient $\boldsymbol{W}$ across all dimensions using the following equation:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{9.1}$$

In > 1 dimensions, our gradient becomes a *vector of partial derivatives*. The problem with this equation is that:

1. It's an *approximation* to the gradient.
2. It's painfully slow.

In practice, we use the *analytic gradient* instead. The method is exact and fast, but extremely challenging to implement due to partial derivatives and multi-variable calculus. Full derivation of the multivariable calculus used to justify gradient descent is outside the scope of this book. If you are interested in learning more about numeric and analytic gradients, I would suggest this lecture by Zibulevsky [80], Andrew Ng's cs229 machine learning notes [81], as well as the cs231n notes [82].

For the sake of this discussion, simply internalize what gradient descent is: attempting to optimize our parameters for low loss and high classification accuracy via an iterative process of taking a step in the direction that minimizes loss.

9.1.3 Treat It Like a Convex Problem (Even if It's Not)

Using the a bowl in Figure 9.1 (*right*) as a visualization of the loss landscape also allows us to draw an important conclusion in modern day neural networks – **we are treating the loss landscape as a convex problem, *even if it's not*.** If some function F is convex, then all local minima are also global minima. This idea fits the visualization of the bowl nicely. Our optimization algorithm simply has to strap on a pair of skis at the top of the bowl, then slowly ride down the gradient until we reach the bottom.

The issue is that nearly all problems we apply neural networks and deep learning algorithms to are *not* neat, convex functions. Instead, inside this bowl we'll find spike-like peaks, valleys that are more akin to canyons, steep dropoffs, and even slots where loss drops dramatically only to sharply rise again.

Given the non-convex nature of our datasets, why do we apply gradient descent? The answer is simple: *because it does a good enough job*. To quote Goodfellow et al. [10]:

> *"[An] optimization algorithm may not be guaranteed to arrive at even a local minimum in a reasonable amount of time, but it often finds a very low value of the [loss] function quickly enough to be useful."*

We can set the high expectation of finding a local/global minimum when training a deep learning network, but this expectation rarely aligns with reality. Instead, we end up finding a region of low loss – *this area may not even be a local minimum*, but in practice, it turns out that this is **good enough**.

9.1.4 The Bias Trick

Before we move on to implementing gradient descent, I want to take the time to discuss a technique called the "bias trick", a method of combining our weight matrix $\boldsymbol{W}$ and bias vector $\boldsymbol{b}$ into a *single* parameter. Recall from our previous decisions that our scoring function is defined as:

$$f(x_i, W, b) = Wx_i + b \tag{9.2}$$

It's often tedious to keep track of two separate variables, both in terms of explanation *and* implementation – to avoid situation this entirely, we can combine $\boldsymbol{W}$ and $\boldsymbol{b}$ together. To combine both the bias and weight matrix, we add an extra dimension (i.e., column) to our input data $\boldsymbol{X}$ that holds a constant 1 – this is our bias dimension.

Typically we either append the new dimension to each individual x_i as the first dimension or the last dimension. In reality, it doesn't matter. We can choose any arbitrary location to insert a

column of ones into our design matrix, as long as it exists. Doing so allows us to rewrite our scoring function via a single matrix multiply:

$$f(x_i, W) = Wx_i \tag{9.3}$$

Again, we are allowed to omit the b term here as it is *embedded* into our weight matrix.

In the context of our previous examples in the "Animals" dataset, we've worked with $32 \times 32 \times 3$ images with a total of 3,072 pixels. Each x_i is represented by a vector of $[3072 \times 1]$. Adding in a dimension with a constant value of one now expands the vector to be $[3073 \times 1]$. Similarly, combining both the bias and weight matrix also expands our weight matrix $\boldsymbol{W}$ to be $[3 \times 3073]$ rather than $[3 \times 3072]$. In this way, we can treat the bias as a *learnable parameter within the weight matrix* that we don't have to explicitly keep track of in a separate variable.

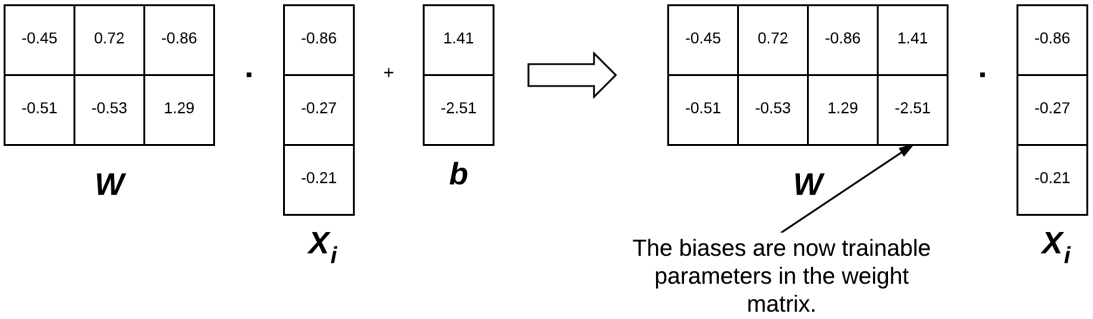


Figure 9.3: **Left:** Normally we treat the weight matrix and bias vector as two separate parameters. **Right:** However, we can actually *embed* the bias vector into the weight matrix (thereby making it a trainable parameter *directly inside* the weight matrix by initializing our weight matrix with an extra column of ones.

To visualize the bias trick, consider Figure 9.3 (*left*) where we *separate* the weight matrix and bias. Up until now, this figure depicts is how we have thought of our scoring function. But instead, we can *combine* the $\boldsymbol{W}$ and $\boldsymbol{b}$ together, provided that we insert a new column into every x_i where every entry is one (Figure 9.3, *right*). Applying the bias trick allows us to learn only a single matrix of weights, hence why we tend to prefer this method for implementation. For all future examples in this book, whenever I mention $\boldsymbol{W}$, assume that the bias vector $\boldsymbol{b}$ is implicitly included in the weight matrix as well.

9.1.5 Pseudocode for Gradient Descent

Below I have included Python-like pseudocode for the standard, vanilla gradient descent algorithm (pseudocode inspired by cs231n slides [83]):

```
1 while True:
2     Wgradient = evaluate_gradient(loss, data, W)
3     W += -alpha * Wgradient
```

This pseudocode is what *all* variations of gradient descent are built off of. We start off on **Line 1** by looping until some condition is met, typically either:

1. A specified number of epochs has passed (meaning our learning algorithm has "seen" each of the training data points N times).

2. Our loss has become *sufficiently low* or training accuracy *satisfactory high*.
3. Loss has not improved in M subsequent epochs.

Line 2 then calls a function named `evaluate_gradient`. This function requires three parameters:

1. `loss`: A function used to compute the loss over our current parameters `W` and input `data`.
2. `data`: Our training data where each training sample is represented by an image (or feature vector).
3. `W`: Our actual weight matrix that we are optimizing over. Our goal is to apply gradient descent to find a `W` that yields minimal loss.

The `evaluate_gradient` function returns a vector that is K-dimensional, where K is the number of dimensions in our image/feature vector. The `Wgradient` variable is the actual gradient, where we have a gradient entry for each dimension.

We then apply *gradient descent* on **Line 3**. We multiply our `Wgradient` by `alpha` (α), which is our *learning rate*. **The learning rate controls the size of our step.**

In practice, you'll spend *a lot* of time finding an optimal value of α – it is *by far* the most important parameter in your model. If α is too large, you'll spend all of your time bounding around the loss landscape, never actually "descending" to the bottom of the basin (unless your random bouncing takes you there by pure luck). Conversely, if α is too small, then it will take *many* (perhaps prohibitively many) iterations to reach the bottom of the basin. Finding the optimal value of α will cause you many headaches – and you'll send a consider amount of your time trying to find an optimal value for this variable for your model and dataset.

9.1.6 Implementing Basic Gradient Descent in Python

Now that we know the basics of gradient descent, let's implement it in Python and use it to classify some data. Open up a new file, name it `gradient_descent.py`, and insert the following code:

```
1  # import the necessary packages
2  from sklearn.model_selection import train_test_split
3  from sklearn.metrics import classification_report
4  from sklearn.datasets import make_blobs
5  import matplotlib.pyplot as plt
6  import numpy as np
7  import argparse
8
9  def sigmoid_activation(x):
10     # compute the sigmoid activation value for a given input
11     return 1.0 / (1 + np.exp(-x))
```

Lines 2-7 import our required Python packages. We have seen all of these imports before, with the exception of `make_blobs`, a function used to create "blobs" of normally distributed data points – this is a handy function when testing or implementing our own models from scratch.

We then define the `sigmoid_activation` function on **Line 9**. When plotted this function will resemble an "S"-shaped curve (Figure 9.4). We call it an ***activation function*** because the function will "activate" and fire "ON" (output value > 0.5) or "OFF" (output value $<= 0.5$) based on the inputs `x`.

We can define this relationship via the `predict` method below:

```
13 def predict(X, W):
14     # take the dot product between our features and weight matrix
15     preds = sigmoid_activation(X.dot(W))
```

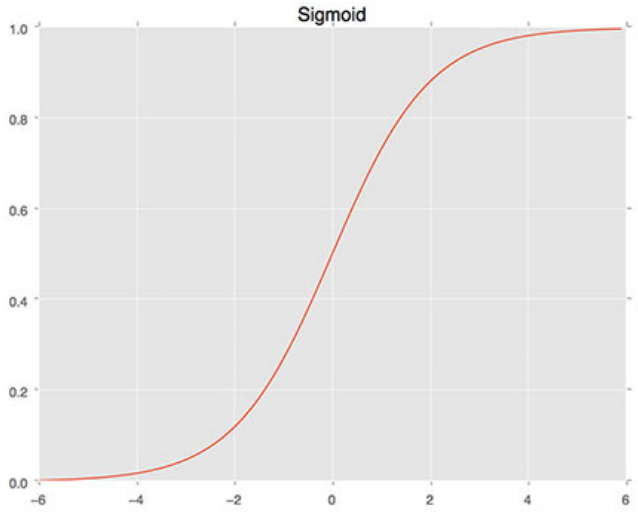


Figure 9.4: The sigmoid activation function. This function is centered at $x = 0.5, y = 0.5$. The function saturates at the tails.

```
16
17      # apply a step function to threshold the outputs to binary
18      # class labels
19      preds[preds <= 0.5] = 0
20      preds[preds > 0] = 1
21
22      # return the predictions
23      return preds
```

Given a set of input data points `X` and weights `W`, we call the `sigmoid_activation` function on them to obtain a set of predictions (**Line 15**). We then threshold the predictions: any prediction with a value $<= 0.5$ is set to `0` while any prediction with a value > 0.5 is set to `1` (**Lines 19 and 20**). The predictions are then returned to the calling function on **Line 23**.

While there are other (better) alternatives to the sigmoid activation function, it makes for an excellent starting point in our discussion of neural networks, deep learning, and gradient-based optimization. I'll be discussing other activation functions in Chapter 10 of the *Starter Bundle* and Chapter 7 of the *Practitioner Bundle*, but for the time being, simply keep in mind that the sigmoid is a non-linear activation function that we can use to threshold our predictions.

Next, let's parse our command line arguments:

```
25  # construct the argument parse and parse the arguments
26  ap = argparse.ArgumentParser()
27  ap.add_argument("-e", "--epochs", type=float, default=100,
28      help="# of epochs")
29  ap.add_argument("-a", "--alpha", type=float, default=0.01,
30      help="learning rate")
31  args = vars(ap.parse_args())
```

We can provide two (optional) command line arguments to our script:

- `--epochs`: The number of epochs that we'll use when training our classifier using gradient descent.

- `--alpha`: The *learning rate* for the gradient descent. We typically see 0.1, 0.01, and 0.001 as initial learning rate values, but again, this is a hyperparameter you'll need to tune for your own classification problems.

Now that our command line arguments are parsed, let's generate some data to classify:

```
33 # generate a 2-class classification problem with 1,000 data points,
34 # where each data point is a 2D feature vector
35 (X, y) = make_blobs(n_samples=1000, n_features=2, centers=2,
36      cluster_std=1.5, random_state=1)
37 y = y.reshape((y.shape[0], 1))
38
39 # insert a column of 1's as the last entry in the feature
40 # matrix -- this little trick allows us to treat the bias
41 # as a trainable parameter within the weight matrix
42 X = np.c_[X, np.ones((X.shape[0]))]
43
44 # partition the data into training and testing splits using 50% of
45 # the data for training and the remaining 50% for testing
46 (trainX, testX, trainY, testY) = train_test_split(X, y,
47      test_size=0.5, random_state=42)
```

On **Line 35** we make a call to `make_blobs` which generates 1,000 data points separated into two classes. These data points are 2D, implying that the "feature vectors" are of length 2. The labels for each of these data points are either `0` or `1`. Our goal is to train a classifier that correctly predicts the class label for each data point.

Line 42 applies the "bias trick" (detailed above) that allows us to skip *explicitly* keeping track of our bias vector $\boldsymbol{b}$, by inserting a brand new column of 1s as the last entry in our design matrix `X`. Adding a column containing a constant value across *all* feature vectors allows us to treat our bias as a *trainable parameter* ***within*** the weight matrix $\boldsymbol{W}$ rather than as an entirely separate variable.

Once we have inserted the column of ones, we partition the data into our training and testing splits on **Lines 46 and 47**, using 50% of the data for training and 50% for testing.

Our next code block handles randomly initializing our weight matrix using a uniform distribution such that it has the same number of dimensions as our input features (including the bias):

```
49 # initialize our weight matrix and list of losses
50 print("[INFO] training...")
51 W = np.random.randn(X.shape[1], 1)
52 losses = []
```

You might also see both *zero* and *one* weight initialization, but as we'll find out later in this book, good initialization is ***critical*** to training a neural network in a reasonable amount of time, so random initialization along with simple heuristics win out in the vast majority of circumstances [84].

Line 52 initializes a list to keep track of our losses after each epoch. At the end of your Python script, we'll plot the loss (which should ideally decrease over time).

All of our variables are now initialized, so we can move on to the actual training and gradient descent procedure:

```
54 # loop over the desired number of epochs
55 for epoch in np.arange(0, args["epochs"]):
```

```
56      # take the dot product between our features `X` and the weight
57      # matrix `W`, then pass this value through our sigmoid activation
58      # function, thereby giving us our predictions on the dataset
59      preds = sigmoid_activation(trainX.dot(W))
60
61      # now that we have our predictions, we need to determine the
62      # `error`, which is the difference between our predictions and
63      # the true values
64      error = preds - trainY
65      loss = np.sum(error ** 2)
66      losses.append(loss)
```

On **Line 55** we start looping over the supplied number of `--epochs`. By default, we'll allow the training procedure to "see" each of the training points a total of 100 times (thus, 100 epochs).

Line 59 takes the dot product between our *entire* training set `trainX` and our weight matrix `W`. The output of this dot product is fed through the sigmoid activation function, yielding our predictions.

Given our predictions, the next step is to determine the "error" of the predictions, or more simply, the difference between our *predictions* and the *true values* (**Line 64**). **Line 65** computes the least squares error over our predictions, a simple loss typically used for binary classification problems. The goal of this training procedure is to minimize our least squares error. We append this `loss` to our `losses` list on **Line 66**, so we can later plot the loss over time.

Now that we have our `error`, we can compute the `gradient` and then use it to update our weight matrix `W`:

```
68      # the gradient descent update is the dot product between our
69      # features and the error of the predictions
70      gradient = trainX.T.dot(error)
71
72      # in the update stage, all we need to do is "nudge" the weight
73      # matrix in the negative direction of the gradient (hence the
74      # term "gradient descent" by taking a small step towards a set
75      # of "more optimal" parameters
76      W += -args["alpha"] * gradient
77
78      # check to see if an update should be displayed
79      if epoch == 0 or (epoch + 1) % 5 == 0:
80          print("[INFO] epoch={}, loss={:.7f}".format(int(epoch + 1),
81              loss))
```

Line 70 handles computing the gradient, which is the dot product between our data points `X` and the `error`.

Line 76 is the most critical step in our algorithm and where the actual gradient descent takes place. Here we update our weight matrix `W` by taking a step in the negative direction of the gradient, thereby allowing us to move towards the bottom of the basin of the loss landscape (hence the term, *gradient descent*). After updating our weight matrix, we check to see if an update should be displayed to our terminal (**Lines 79-81**) and then keep looping until the desired number of epochs has been met – gradient descent is thus an *iterative algorithm*.

Our classifier is now trained. The next step is evaluation:

```
83  # evaluate our model
84  print("[INFO] evaluating...")
```

```
85 preds = predict(testX, W)
86 print(classification_report(testY, preds))
```

To actually make predictions using our weight matrix `W`, we call the `predict` method on `testX` and `W` on **Line 85**. Given the predictions, we display a nicely formatted classification report to our terminal on **Line 86**.

Our last code block handles plotting (1) the *testing data* so we can visualize the dataset we are trying to classify and (2) our loss over time:

```
88 # plot the (testing) classification data
89 plt.style.use("ggplot")
90 plt.figure()
91 plt.title("Data")
92 plt.scatter(testX[:, 0], testX[:, 1], marker="o", c=testY, s=30)
93
94 # construct a figure that plots the loss over time
95 plt.style.use("ggplot")
96 plt.figure()
97 plt.plot(np.arange(0, args["epochs"]), losses)
98 plt.title("Training Loss")
99 plt.xlabel("Epoch #")
100 plt.ylabel("Loss")
101 plt.show()
```

9.1.7 Simple Gradient Descent Results

To execute our script, simply issue the following command:

```
$ python gradient_descent.py
[INFO] training...
[INFO] epoch=1, loss=486.5895513
[INFO] epoch=5, loss=11.1087812
[INFO] epoch=10, loss=9.1312984
[INFO] epoch=15, loss=7.0049498
[INFO] epoch=20, loss=6.9914949
[INFO] epoch=25, loss=6.9382765
[INFO] epoch=30, loss=5.8285461
[INFO] epoch=35, loss=4.1750536
[INFO] epoch=40, loss=2.7319634
[INFO] epoch=45, loss=1.3891531
[INFO] epoch=50, loss=1.0787992
[INFO] epoch=55, loss=0.8927193
[INFO] epoch=60, loss=0.6001450
[INFO] epoch=65, loss=0.3200953
[INFO] epoch=70, loss=0.1651333
[INFO] epoch=75, loss=0.0941329
[INFO] epoch=80, loss=0.0602669
[INFO] epoch=85, loss=0.0424516
[INFO] epoch=90, loss=0.0321485
[INFO] epoch=95, loss=0.0256970
[INFO] epoch=100, loss=0.0213877
```

As we can see from Figure 9.5 (*left*), our dataset is clearly linear separable (i.e., we can draw a line that separates the two classes of data). Our loss also drops dramatically, starting out very high

and then quickly dropping (*right*). We can see just how quickly the loss drops by investigating the terminal output above. Notice how the loss is initially > 400 but drops to ≈ 1.0 by epoch 50. By the time training terminates by epoch 100, our loss has dropped by an order of magnitude to 0.02.

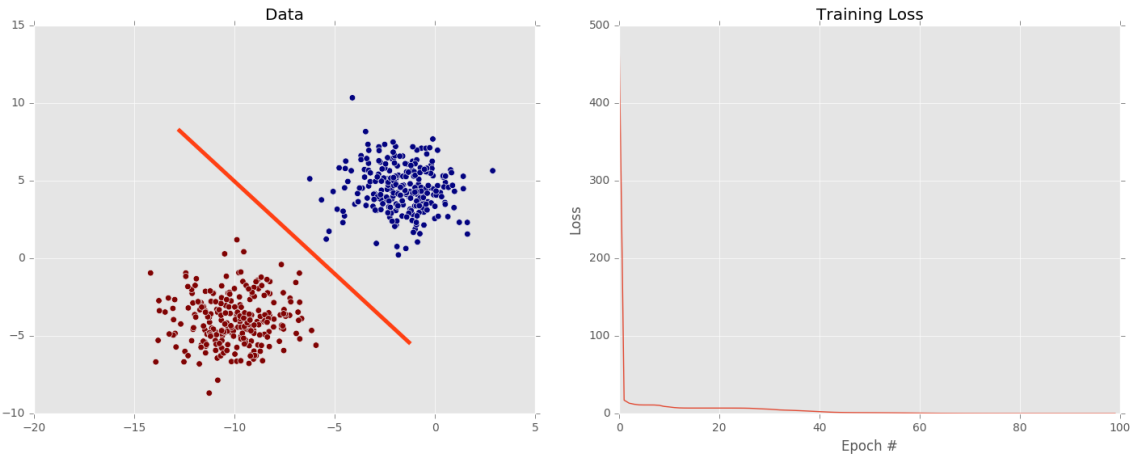


Figure 9.5: **Left:** The input dataset that we are trying to classify into two sets: red and blue. This dataset is clearly linearly separable as we can draw a single line that neatly divides the dataset into two classes. **Right:** Learning a set of parameters to classify our dataset via gradient descent. Loss starts very high but rapidly drops to nearly zero.

This plot validates that our weight matrix is being updated in a manner that allows the classifier to learn from the training data. However, based on the rest of our terminal output, it seems that our classifier misclassified a handful of data points (< 5 of them):

```
[INFO] evaluating...
             precision    recall  f1-score   support

          0       1.00      0.99      1.00       250
          1       0.99      1.00      1.00       250

avg / total       1.00      1.00      1.00       500
```

Notice how the zero class is classified correctly 100% of the time, but the one class is classified correctly only 99% of the time. The reason for this discrepancy is because vanilla gradient descent only performs a weight update *once* for every epoch – in this example, we trained our model for 100 epochs, so only 100 updates took place. Depending on the initialization of the weight matrix and the size of the learning rate, it's possible that we might not be able to learn a model that can separate the points (even though they are linearly separable).

In fact, subsequent runs of this script may reveal that *both* classes can be classified correctly 100% of the time – the result is dependent on the initial values `W` takes on. To verify this result yourself, run the `gradient_descent.py` script multiple times.

For simple gradient descent, you are better off training for *more epochs* with a *smaller learning rate* to help overcome this issue. However, as we'll see in the next section, a variant of gradient descent called *Stochastic Gradient Descent* performs a weight update for *every batch* of training data, implying there are *multiple* weight updates per epoch. This approach leads to a faster, more stable convergence.

9.2 Stochastic Gradient Descent (SGD)

In the previous section, we discussed gradient descent, a first-order optimization algorithm that can be used to learn a set of classifier weights for parameterized learning. However, this "vanilla" implementation of gradient descent can be prohibitively slow to run on large datasets – in fact, it can even be considered *computational wasteful*.

Instead, we should apply **Stochastic Gradient Descent (SGD)**, a simple modification to the standard gradient descent algorithm that *computes the gradient* and *updates the weight matrix* $\boldsymbol{W}$ on **small batches of training data**, rather than the entire training set. While this modification leads to "more noisy" updates, it also allows us to take *more steps along the gradient* (one step per each batch versus one step per epoch), ultimately leading to faster convergence and no negative affects to loss and classification accuracy.

SGD is arguably ***the most important algorithm*** when it comes to training deep neural networks. Even though the original incarnation of SGD was introduced over 57 years ago [85], it is *still* the engine that enables us to train large networks to learn patterns from data points. Above all other algorithms covered in this book, take the time to understand SGD.

9.2.1 Mini-batch SGD

Reviewing the vanilla gradient descent algorithm, it should be (somewhat) obvious that the method will run *very slowly* on large datasets. The reason for this slowness is because each iteration of gradient descent requires us to compute a prediction for each training point in our training data *before* we are allowed to update our weight matrix. For image datasets such as ImageNet were we have over *1.2 million* training images, this computation can take a long time.

It also turns out that computing predictions for *every* training point before taking a step along our weight matrix is computationally wasteful and does little to help our model coverage.

Instead, what we should do is *batch* our updates. We can update the pseudocode to transform vanilla gradient descent to become SGD by adding an extra function call:

```
while True:
    batch = next_training_batch(data, 256)
    Wgradient = evaluate_gradient(loss, batch, W)
    W += -alpha * Wgradient
```

The only difference between vanilla gradient descent and SGD is the addition of the `next_training_batch` function. Instead of computing our gradient over the *entire* `data` set, we instead sample our data, yielding a `batch`. We evaluate the gradient on the `batch`, and update our weight matrix `W`. From an implementation perspective, we also try to randomize our training samples *before* applying SGD since the algorithm is sensitive to batches.

After looking at the pseudocode for SGD, you'll immediately notice an introduction of a new parameter: ***the batch size***. In a "purist" implementation of SGD, your mini-batch size would be 1, implying that we would randomly sample *one* data point from the training set, compute the gradient, and update our parameters. However, we often use mini-batches that are > 1. Typical batch sizes include 32, 64, 128, and 256.

So, why bother using batch sizes > 1? To start, batch sizes > 1 help reduce variance in the parameter update (http://pyimg.co/pd5w0), leading to a more stable convergence. Secondly, powers of two are often desirable for batch sizes as they allow internal linear algebra optimization libraries to be more efficient.

In general, the mini-batch size is not a hyperparameter you should worry too much about [57]. If you're using a GPU to train your neural network, you determine how many training examples will fit into your GPU and then use the nearest power of two as the batch size such that the batch

will fit on the GPU. For CPU training, you typically use one of the batch sizes listed above to ensure you reap the benefits of linear algebra optimization libraries.

9.2.2 Implementing Mini-batch SGD

Let's go ahead and implement SGD and see how it differs from standard vanilla gradient descent. Open up a new file, name it `sgd.py`, and insert the following code:

```
1  # import the necessary packages
2  from sklearn.model_selection import train_test_split
3  from sklearn.metrics import classification_report
4  from sklearn.datasets import make_blobs
5  import matplotlib.pyplot as plt
6  import numpy as np
7  import argparse
8
9  def sigmoid_activation(x):
10     # compute the sigmoid activation value for a given input
11     return 1.0 / (1 + np.exp(-x))
```

Lines 2-7* import our required Python packages, exactly the same as the `gradient_descent.py` example earlier in this chapter. **Lines 9-11** define the `sigmoid_activation` function, which is also identical to the previous version of gradient descent.

In fact, the `predict` method doesn't change either:

```
13 def predict(X, W):
14     # take the dot product between our features and weight matrix
15     preds = sigmoid_activation(X.dot(W))
16
17     # apply a step function to threshold the outputs to binary
18     # class labels
19     preds[preds <= 0.5] = 0
20     preds[preds > 0] = 1
21
22     # return the predictions
23     return preds
```

However, what *does* change is the addition of the `next_batch` function:

```
25 def next_batch(X, y, batchSize):
26     # loop over our dataset `X` in mini-batches, yielding a tuple of
27     # the current batched data and labels
28     for i in np.arange(0, X.shape[0], batchSize):
29         yield (X[i:i + batchSize], y[i:i + batchSize])
```

The `next_batch` method requires three parameters:

1. `X`: Our training dataset of feature vectors/raw image pixel intensities.
2. `y`: The class labels associated with each of the training data points.
3. `batchSize`: The size of each mini-batch that will be returned.

Lines 28 and 29 then loop over the training examples, yielding subsets of both `X` and `y` as mini-batches.

Next, we can parse our command line arguments:

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-e", "--epochs", type=float, default=100,
	help="# of epochs")
ap.add_argument("-a", "--alpha", type=float, default=0.01,
	help="learning rate")
ap.add_argument("-b", "--batch-size", type=int, default=32,
	help="size of SGD mini-batches")
args = vars(ap.parse_args())
```

We have already reviewed both the `--epochs` (number of epochs) and `--alpha` (learning rate) switch from the vanilla gradient descent example – but also notice we are introducing a third switch: `--batch-size`, which as the name indicates is the size of each of our mini-batches. We'll default this value to be 32 data points per mini-batch.

Our next code block handles generating our 2-class classification problem with 1,000 data points, adding the bias column, and then performing the training and testing split:

```
  # generate a 2-class classification problem with 1,000 data points,
# where each data point is a 2D feature vector
(X, y) = make_blobs(n_samples=1000, n_features=2, centers=2,
	cluster_std=1.5, random_state=1)
y = y.reshape((y.shape[0], 1))

# insert a column of 1's as the last entry in the feature
# matrix -- this little trick allows us to treat the bias
# as a trainable parameter within the weight matrix
X = np.c_[X, np.ones((X.shape[0]))]

# partition the data into training and testing splits using 50% of
# the data for training and the remaining 50% for testing
(trainX, testX, trainY, testY) = train_test_split(X, y,
	test_size=0.5, random_state=42)
```

We'll then initialize our weight matrix and losses just like in the previous example:

```
# initialize our weight matrix and list of losses
print("[INFO] training...")
W = np.random.randn(X.shape[1], 1)
losses = []
```

The *real* change comes next where we loop over the desired number of epochs, sampling mini-batches along the way:

```
# loop over the desired number of epochs
for epoch in np.arange(0, args["epochs"]):
	# initialize the total loss for the epoch
	epochLoss = []

	# loop over our data in batches
	for (batchX, batchY) in next_batch(X, y, args["batch_size"]):
		# take the dot product between our current batch of features
```

```
70              # and the weight matrix, then pass this value through our
71              # activation function
72              preds = sigmoid_activation(batchX.dot(W))
73
74              # now that we have our predictions, we need to determine the
75              # ‘error‘, which is the difference between our predictions
76              # and the true values
77              error = preds - batchY
78              epochLoss.append(np.sum(error ** 2))
```

On **Line 63** we start looping over the supplied number of `--epochs`. We then loop over our training data in batches on **Line 68**. For each batch, we compute the dot product between the batch and `W`, then pass the result through the sigmoid activation function to obtain our predictions. We compute the least square error for the batch on **Line 77** and use this value to update our `epochLoss` on **Line 78**.

Now that we have the `error`, we can compute the gradient descent update, which is the dot product between the current batch data points and the error on the batch:

```
80              # the gradient descent update is the dot product between our
81              # current batch and the error on the batch
82              gradient = batchX.T.dot(error)
83
84              # in the update stage, all we need to do is "nudge" the
85              # weight matrix in the negative direction of the gradient
86              # (hence the term "gradient descent") by taking a small step
87              # towards a set of "more optimal" parameters
88              W += -args["alpha"] * gradient
```

Line 88 handles updating our weight matrix based on the gradient, scaled by our learning rate `--alpha`. Notice how the weight update stage takes place *inside* the batch loop – this implies there are *multiple weight updates per epoch*.

We can then update our loss history by taking the average across all batches in the epoch and then displaying an update to our terminal if necessary:

```
90          # update our loss history by taking the average loss across all
91          # batches
92          loss = np.average(epochLoss)
93          losses.append(loss)
94
95          # check to see if an update should be displayed
96          if epoch == 0 or (epoch + 1) % 5 == 0:
97              print("[INFO] epoch={}, loss={:.7f}".format(int(epoch + 1),
98                  loss))
```

Evaluating our classifier is done in the same way as in vanilla gradient descent – simply call `predict` on the `testX` data using our learned `W` weight matrix:

```
100 # evaluate our model
101 print("[INFO] evaluating...")
102 preds = predict(testX, W)
103 print(classification_report(testY, preds))
```

We'll end our script by plotting the testing classification data and along with the loss per epoch:

```
# plot the (testing) classification data
plt.style.use("ggplot")
plt.figure()
plt.title("Data")
plt.scatter(testX[:, 0], testX[:, 1], marker="o", c=testY, s=30)

# construct a figure that plots the loss over time
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, args["epochs"]), losses)
plt.title("Training Loss")
plt.xlabel("Epoch #")
plt.ylabel("Loss")
plt.show()
```

9.2.3 SGD Results

To visualize the results from our implementation, just execute the following command:

```
$ python sgd.py
[INFO] training...
[INFO] epoch=1, loss=0.3701232
[INFO] epoch=5, loss=0.0195247
[INFO] epoch=10, loss=0.0142936
[INFO] epoch=15, loss=0.0118625
[INFO] epoch=20, loss=0.0103219
[INFO] epoch=25, loss=0.0092114
[INFO] epoch=30, loss=0.0083527
[INFO] epoch=35, loss=0.0076589
[INFO] epoch=40, loss=0.0070813
[INFO] epoch=45, loss=0.0065899
[INFO] epoch=50, loss=0.0061647
[INFO] epoch=55, loss=0.0057920
[INFO] epoch=60, loss=0.0054620
[INFO] epoch=65, loss=0.0051670
[INFO] epoch=70, loss=0.0049015
[INFO] epoch=75, loss=0.0046611
[INFO] epoch=80, loss=0.0044421
[INFO] epoch=85, loss=0.0042416
[INFO] epoch=90, loss=0.0040575
[INFO] epoch=95, loss=0.0038875
[INFO] epoch=100, loss=0.0037303
[INFO] evaluating...
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       250
           1       1.00      1.00      1.00       250

   avg / total       1.00      1.00      1.00       50
```

We'll be using the same "blob" dataset as in Figure 9.5 (*left*) above for classification so we can compare our SGD results to vanilla gradient descent. Furthermore, SGD example uses the same

learning rate (0.1) and the same number of epochs (100) as vanilla gradient descent. However, notice how much *smoother* our loss curve is in Figure 9.6.

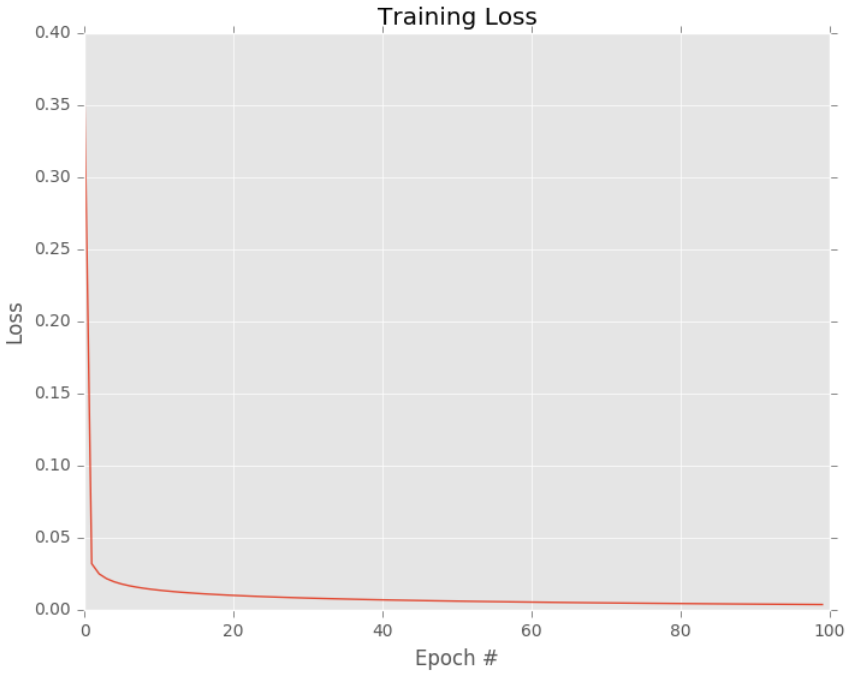


Figure 9.6: Applying Stochastic Gradient Descent to our dataset of red and blue data points. Using SGD, our learning curve is *much* smoother. Furthermore, we are able to obtain an order of magnitude lower loss by the end of the 100th epoch (as compared to standard, vanilla gradient descent).

Investigating the actual loss values at the end of the 100th epoch, you'll notice that loss obtained by SGD is ***an order of magnitude lower*** than vanilla gradient descent (0.003 vs 0.021, respectively). This difference is due to the multiple weight updates per epoch, giving our model more chances to learn from the updates made to the weight matrix. This effect is even more pronounced on large datasets, such as ImageNet where we have millions of training examples and small, incremental updates in our parameters can lead to a low loss (but not necessarily optimal) solution.

9.3 Extensions to SGD

There are two primary extensions that you'll encounter to SGD in practice. The first is momentum [86], a method used to accelerate SGD, enabling it to learn faster by focusing on dimensions whose gradient point in the same direction. The second method is Nesterov acceleration [87], an extension to standard momentum.

9.3.1 Momentum

Consider your favorite childhood playground where you spent days rolling down a hill, covering yourself in grass and dirt (much to your mother's chagrin). As you travel down the hill, you build up more and more momentum, which in turn carries you faster down the hill.

Momentum applied to SGD has the same effect – our goal is to build upon the standard weight update to include a momentum term, thereby allowing our model to obtain lower loss (and higher

accuracy) in less epochs. The momentum term should, therefore, *increase* the strength of updates for dimensions who gradients point in the same direction and then *decrease* the strength of updates for dimensions who gradients switch directions [86, 88].

Our previous weight update rule simply included the scaling the gradient by our learning rate:

$$W = W - \alpha \nabla_W f(W) \tag{9.4}$$

We now introduce the momentum term V, scaled by γ:

$$V = \gamma V - \alpha \nabla_W f(W) \qquad W = W + V \tag{9.5}$$

The momentum term γ is commonly set to 0.9; although another common practice is to set γ to 0.5 until learning stabilizes and then increase it to 0.9 – it is extremely rare to see momentum < 0.5. For a more detailed review of momentum, please refer to Sutton [89] and Qian [86].

9.3.2 Nesterov's Acceleration

Let's suppose that you are back on your childhood playground, rolling down the hill. You've built up momentum and are moving quite fast – but there's a problem. At the bottom of the hill is the brick wall of your school, one that you would like to avoid hitting at full speed.

The same thought can be applied to SGD. If we build up too much momentum, we may overshoot a local minimum and keep on rolling. Therefore, it would be advantageous to have a smarter roll, one that knows when to slow down, which is where Nesterov accelerated gradient [87] comes in.

Nesterov acceleration can be conceptualized as a corrective update to the momentum which lets us obtain an approximate idea of where our parameters will be after the update. Looking at Hinton's *Overview of mini-batch gradient descent* slides [90], we can see a nice visualization of Nesterov acceleration (Figure 9.7).

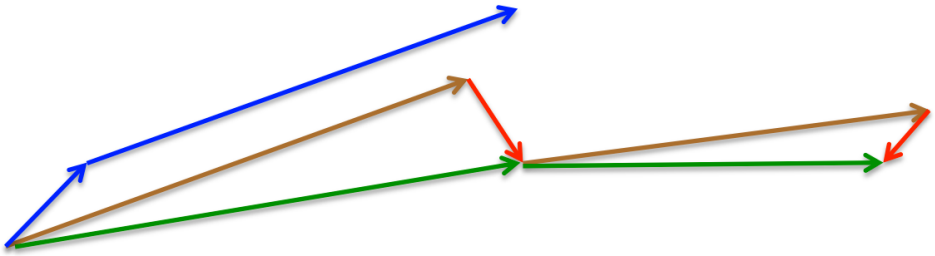


Figure 9.7: A graphical depecition of Nesterov acceleration. First, we make a big jump in the direction of the previous gradient, then measure the gradient where we ended up and make the correction.

Using standard momentum, we compute the gradient (small blue vector) and then take a big jump in the in the direction of the gradient (large blue vector). Under Nesterov acceleration we would first make a big jump in the direction of our *previous* gradient (brown vector), measure the gradient, and then make a *correction* (red vector) – the green vector is the final corrected update by Nesterov acceleration (paraphrased from Ruder [88]).

A thorough theoretical and mathematical treatment of Nesterov acceleration are outside the scope of this book. For those interested in studying Nesterov acceleration in more detail, please refer to Ruder [88], Bengio [91], and Sutskever [92].

9.3.3 Anecdotal Recommendations

Momentum is an important term that can increase the convergence of our model; we tend not to worry with this hyperparameter as much, as compared to our learning rate and regularization penalty (discussed in the next section), which are *by far* the most important knobs to tweak.

My personal rule of thumb is that whenever using SGD, also apply momentum. In most cases, you can set it (and leave it) and 0.9 although Karpathy [93] suggests starting at 0.5 and increasing it to larger values as your epochs increase.

As for Nesterov acceleration, I tend to use it on smaller datasets, but for larger datasets (such as ImageNet), I almost always avoid it. While Nesterov acceleration has sound theoretical guarantees, all major publications trained on ImageNet (e.x., AlexNet [94], VGGNet [95], ResNet [96], Inception [97], etc.) use SGD with momentum – *not a single paper from this seminal group utilizes Nesterov acceleration*.

My personal experience has lead me to find that when training deep networks on large datasets, SGD is easier to work with when using momentum and *leaving out* Nesterov acceleration. Smaller datasets, on the other hand, tend to enjoy the benefits of Nesterov acceleration. However, keep in mind that this is my anecdotal opinion and that your mileage may vary.

9.4 Regularization

> *"Many strategies used in machine learning are explicitly designed to reduce the test error, possibly at the expense of increased training error. These strategies are collectively known as regularization."* – Goodfellow et al. [10]

In earlier sections of this chapter, we discussed two important loss functions: Multi-class SVM loss and cross-entropy loss. We then discussed gradient descent and how a network can actually learn by updating the weight parameters of a model. While our loss function allows us to determine how well (or poorly) our set of parameters are performing on a given classification task, the loss function itself does not take into account how the weight matrix "looks".

What do I mean by "looks"? Well, keep in mind that we are working in a real-valued space, thus there are an *infinite set* of parameters that will obtain reasonable classification accuracy on our dataset (for some definition of "reasonable").

How do we go about choosing a set of parameters that help ensure our model generalizes well? Or, at the very least, lessen the effects of overfitting. ***The answer is regularization.*** Second only to your learning rate, regularization is the most important parameter of your model that can you tune.

There are various types of regularization techniques, such as L1 regularization, L2 regularization (commonly called "weight decay"), and Elastic Net [98], that are used by updating the loss function itself, adding an additional parameter to constrain the capacity of the model.

We also have types of regularization that can be *explicitly* added to the network architecture – dropout is the quintessential example of such regularization. We then have *implicit* forms of regularization that are applied during the training process. Examples of implicit regularization include data augmentation and early stopping. Inside this section, we'll mainly be focusing on the parameterized regularization obtained by modifying our loss and update functions.

In Chapter 11 of the *Starter Bundle*, we'll review dropout and then in Chapter 17 we'll discuss overfitting in more depth, as well as how we can use early stopping as a regularizer. Inside the *Practitioner Bundle*, you'll find examples of data augmentation used as regularization.

9.4.1 What Is Regularization and Why Do We Need It?

Regularization helps us control our model capacity, ensuring that our models are better at making (correct) classifications on data points that they were *not* trained on, which we call ***the***

ability to generalize. If we don't apply regularization, our classifiers can easily become too complex and *overfit* to our training data, in which case we lose the ability to generalize to our testing data (and data points *outside* the testing set as well, such as new images in the wild).

However, too much regularization can be a bad thing. We can run the risk of *underfitting*, in which case our model performs poorly on the training data and is not able to model the relationship between the input data and output class labels (because we limited model capacity too much). For example, consider the following plot of points, along with various functions that fit to these points (Figure 9.8).

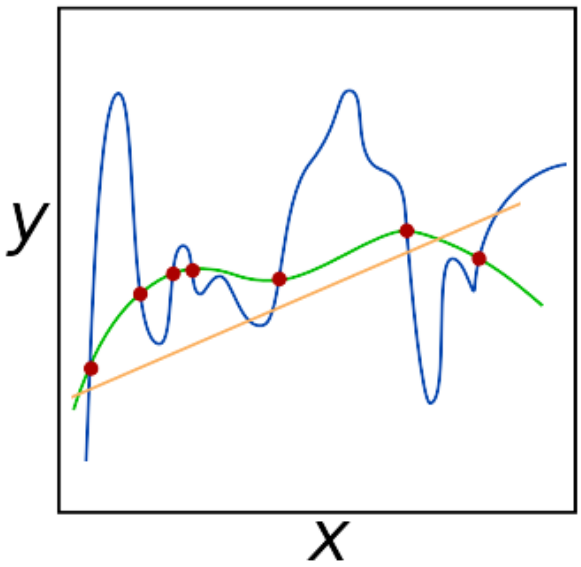


Figure 9.8: An example of underfitting (orange line), overfitting (blue line), and generalizing (green line). Our goal when building deep learning classifiers is to obtain these types of "green functions" that fit our training data nicely, but avoid overfitting. Regularization can help us obtain this type of desired fit.

The orange line is an example of *underfitting* – we are not capturing the relationship between the points. On the other hand, the blue line is an example of *overfitting* – we have too many parameters in our model, and while it hits all points in the dataset, it also wildly varies between the points. It is not a smooth, simple fit that we would prefer. We then have the green function which also hits all points in our dataset, but does so in a much more predictable, simple manner.

The goal of regularization is to obtain these types of "green functions" that fit our training data nicely, but avoid overfitting to our training data (blue) or failing to model the underlying relationship (yellow). We discuss how to monitor training and spot both underfitting and overfitting in Chapter 17; however, for the time being, simply understand that regularization is a critical aspect of machine learning and we use regularization to control model generalization. To understand regularization and the impact it has on our loss function and weight update rule, let's proceed to the next section.

9.4.2 Updating Our Loss and Weight Update To Include Regularization

Let's start with our cross-entropy loss function (Section 8.2.3):

$$L_i = -log(e^{s_{y_i}} / \sum_j e^{s_j}) \tag{9.6}$$

The loss over the entire training set can be written as:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i \tag{9.7}$$

Now, let's say that we have obtained a weight matrix $\boldsymbol{W}$ such that *every data point* in our training set is classified correctly, which means that our loss $L = 0$ for all L_i.

Awesome, we're getting 100% accuracy – but let me ask you a question about this weight matrix – ***is it unique?*** **Or, in other words, are there *better* choices of $\boldsymbol{W}$ that will improve our model's ability to generalize and reduce overfitting?**

If there is such a $\boldsymbol{W}$, how do we know? And how can we incorporate this type of penalty into our loss function? The answer is to define a **regularization penalty**, a function that operates on our weight matrix. The regularization penalty is commonly written as a function, $R(\boldsymbol{W})$. Equation 9.8 below shows the most common regularization penalty, L2 regularization (also called **weight decay**):

$$R(W) = \sum_i \sum_j W_{i,j}^2 \tag{9.8}$$

What is the function doing exactly? In terms of Python code, it's simply taking the sum of squares over an array:

```
penalty = 0

for i in np.arange(0, W.shape[0]):
    for j in np.arange(0, W.shape[1]):
        penalty += (W[i][j] ** 2)
```

What we are doing here is looping over all entries in the matrix and taking the sum of squares. The sum of squares in the L2 regularization penalty discourages large weights in our matrix $\boldsymbol{W}$, preferring smaller ones. Why might we want to discourage large weight values? In short, by penalizing large weights, we can improve the ability to generalize, and thereby reduce overfitting.

Think of it this way – the larger a weight value is, the more influence it has on the output prediction. Dimensions with larger weight values can almost singlehandedly control the output prediction of the classifier (provided the weight value is large enough, of course) which will almost certainly lead to overfitting.

To mitigate affect various dimensions have on our output classifications, we apply regularization, thereby seeking $\boldsymbol{W}$ values that take into account *all* of the dimensions rather than the few with large values. In practice you may find that regularization hurts your training accuracy slightly, but actually *increases your testing accuracy*.

Again, our loss function has the same basic form, only now we add in regularization:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda R(W) \tag{9.9}$$

The first term we have seen before – it is the average loss over all samples in our training set.

The second term is new – *this is our regularization penalty*. The λ variable is a hyperparameter that controls the *amount* or *strength* of the regularization we are applying. In practice, both the learning rate α and the regularization term λ are the hyperparameters that you'll spending the most time tuning.

Expanding cross-entropy loss to include L2 regularization yields the following equation:

$$L = \frac{1}{N}\sum_{i=1}^{N}[-log(e^{s_{y_i}}/\sum_j e^{s_j})] + \lambda \sum_i \sum_j W_{i,j}^2 \tag{9.10}$$

We can also expand Multi-class SVM loss as well:

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}[max(0, s_j - s_{y_i} + 1)] + \lambda \sum_i \sum_j W_{i,j}^2 \tag{9.11}$$

Now, let's take a look at our standard weight update rule:

$$W = W - \alpha \nabla_W f(W) \tag{9.12}$$

This method updates our weights based on the gradient multiple by a learning rate α. Taking into account regularization, the weight update rule becomes:

$$W = W - \alpha \nabla_W f(W) + \lambda R(W) \tag{9.13}$$

Here we are adding a negative linear term to our gradients (i.e., gradient descent), penalizing large weights, with the end goal of making it easier for our model to generalize.

9.4.3 Types of Regularization Techniques

In general, you'll see three common types of regularization there are applied directly to the loss function. The first, we reviewed earlier, L2 regularization (aka "weight decay"):

$$R(W) = \sum_i \sum_j W_{i,j}^2 \tag{9.14}$$

We also have L1 regularization which takes the absolute value rather than the square:

$$R(W) = \sum_i \sum_j |W_{i,j}| \tag{9.15}$$

Elastic Net [98] regularization seeks to combine both L1 and L2 regularization:

$$R(W) = \sum_i \sum_j \beta W_{i,j}^2 + |W_{i,j}| \tag{9.16}$$

Other types of regularization methods exist such as directly modifying the architecture of a network along with how the network is actually trained – we will review these methods in later chapters.

In terms of *which* regularization method you should be using (including none at all), you should treat this choice as a hyperparameter you need to optimize over and perform experiments to determine *if* regularization should be applied, and if so *which method* of regularization, and what the proper value of λ is. For more details on regularization, refer to Chapter 7 of Goodfellow et al. [10], the "Regularization" section from the DeepLearing.net tutorial [99], and the notes from Karpathy's cs231n Neural Networks II lecture [100].

9.4.4 Regularization Applied to Image Classification

To demonstrate regularization in action, let's write some Python code to apply it to our "Animals" dataset. Open up a new file, name it `regularization.py`, and insert the following code:

```
1  # import the necessary packages
2  from sklearn.linear_model import SGDClassifier
3  from sklearn.preprocessing import LabelEncoder
4  from sklearn.model_selection import train_test_split
5  from pyimagesearch.preprocessing import SimplePreprocessor
6  from pyimagesearch.datasets import SimpleDatasetLoader
7  from imutils import paths
8  import argparse
```

Lines 2-8 import our required Python packages. We've seen all of these imports before, except the scikit-learn `SGDClassifier`. As the name of this class suggests, this implementation encapsulates all the concepts we have reviewed in this chapter, including:

- Loss functions
- Number of epochs
- Learning rate
- Regularization terms

Thus making it the perfect example to demonstrate all these concepts in action.

Next, we can parse our command line arguments and grab the list of images from disk:

```
10 # construct the argument parse and parse the arguments
11 ap = argparse.ArgumentParser()
12 ap.add_argument("-d", "--dataset", required=True,
13      help="path to input dataset")
14 args = vars(ap.parse_args())
15
16 # grab the list of image paths
17 print("[INFO] loading images...")
18 imagePaths = list(paths.list_images(args["dataset"]))
```

Given the image paths, we'll resize them to 32×32 pixels, load them from disk into memory, and then flatten them into a 3,072-dim array:

```
20 # initialize the image preprocessor, load the dataset from disk,
21 # and reshape the data matrix
22 sp = SimplePreprocessor(32, 32)
23 sdl = SimpleDatasetLoader(preprocessors=[sp])
24 (data, labels) = sdl.load(imagePaths, verbose=500)
25 data = data.reshape((data.shape[0], 3072))
```

We'll also encode the labels as integers and perform a training testing split, using 75% of the data for training and the remaining 25% for testing:

```
27 # encode the labels as integers
28 le = LabelEncoder()
29 labels = le.fit_transform(labels)
30
31 # partition the data into training and testing splits using 75% of
32 # the data for training and the remaining 25% for testing
33 (trainX, testX, trainY, testY) = train_test_split(data, labels,
34      test_size=0.25, random_state=5)
```

Let's apply a few different types of regularization when training our `SGDClassifier`:

```
36 # loop over our set of regularizers
37 for r in (None, "l1", "l2"):
38      # train a SGD classifier using a softmax loss function and the
39      # specified regularization function for 10 epochs
40      print("[INFO] training model with `{}` penalty".format(r))
41      model = SGDClassifier(loss="log", penalty=r, max_iter=10,
42           learning_rate="constant", eta0=0.01, random_state=42)
43      model.fit(trainX, trainY)
44
45      # evaluate the classifier
46      acc = model.score(testX, testY)
47      print("[INFO] `{}` penalty accuracy: {:.2f}%".format(r,
48           acc * 100))
```

Line 37 loops over our regularizers, including no regularization. We then initialize and train the `SGDClassifier` on **Lines 41-43**.

We'll be using cross-entropy loss, with regularization penalty of `r` and a default λ of 0.0001. We'll use SGD to train the model for 10 epochs with a learning rate of $\alpha = 0.01$. We then evaluate the classifier and display the accuracy results to our screen on **Lines 46-48**.

To see our SGD model trained with various regularization types, just execute the following command:

```
$ python regularization.py --dataset ../datasets/animals
[INFO] loading images...
...
[INFO] training model with `None` penalty
[INFO] `None` penalty accuracy: 50.40%
[INFO] training model with `l1` penalty
[INFO] `l1` penalty accuracy: 52.53%
[INFO] training model with `l2` penalty
[INFO] `l2` penalty accuracy: 55.07%
```

We can see with *no regularization* we obtain an accuracy of **50.40%**. Using L1 regularization our accuracy increases to **52.53%**. L2 regularization obtains the highest accuracy of **55.07%**.

Using different `random_state` values for `train_test_split` will yield different results. The dataset here is too small and the classifier too simplistic to see the full impact of

regularization, so consider this a "worked example". As we continue to work through this book you'll see more advanced uses of regularization that will have dramatic impacts on your accuracy.

Realistically, this example is too small to show all the advantages of applying regularization – for that, we'll have to wait until we start training Convolutional Neural Networks. However, in the meantime simply appreciate that regularization can provide a boost in our testing accuracy and reduce overfitting, *provided we can tune the hyperparameters right*.

9.5 Summary

In this chapter, we popped the hood on deep learning and took a deep dive into the engine that powers modern day neural networks – *gradient descent*. We investigated two types of gradient descent:

1. The standard vanilla flavor.
2. The stochastic version that is more commonly used.

Vanilla gradient descent performs only *one* weight update per epoch, making it very slow (if not impossible) to converge on large datasets. The stochastic version instead applies *multiple* weight updates per epoch by computing the gradient on small mini-batches. By using SGD we can dramatically reduce the time it takes to train a model while also enjoying lower loss and higher accuracy. Typical batch sizes include 32, 64, 128 and 256.

Gradient descent algorithms are controlled via a ***learning rate***: this is by far the most important parameter to tune correctly when training your own models.

If your learning rate is too large, you'll simply bounce around the loss landscape and not actually "learn" any patterns from your data. On the other hand, if your learning rate is too small, it will take a prohibitive number of iterations to reach even a reasonable loss. To get it just right, you'll want to spend the majority of your time tuning the learning rate.

We then discussed ***regularization***, which is defined as *"any method that increases testing accuracy perhaps at the expense of training accuracy"*. Regularization encompasses a broad range of techniques. We specifically focused on regularization methods that are applied to our loss functions and weight update rules, including L1 regularization, L2 regularization, and Elastic Net.

In terms of deep learning and neural networks, you'll commonly see L2 regularization used for image classification – the trick is tuning the λ parameter to include just the right amount of regularization.

At this point, we have a sound foundation of machine learning, but we have yet to investigate neural networks or train a custom neural network from scratch. That will all change in our next chapter where we discuss neural networks, the backpropagation algorithm, and how to train your own neural networks on custom datasets.

10. Neural Network Fundamentals

In this chapter, we'll study the fundamentals of neural networks in depth. We'll start with a discussion of artificial neural networks and how they are inspired by the real-life biological neural networks in our own bodies. From there, we'll review the classic Perceptron algorithm and the role it has played in neural network history.

Building on the Perceptron, we'll also study the *backpropagation algorithm*, the cornerstone of modern neural learning – without backpropagation, we would be unable to efficiently train our networks. We'll also implement backpropagation with Python from scratch, ensuring we understand this important algorithm.

Of course, modern neural network libraries such as Keras already have (highly optimized) backpropagation algorithms built-in. Implementing backpropagation by hand each time we wished to train a neural network would be like coding a linked list or hash table data structure from scratch each time we worked on a general purpose programming problem – not only is it unrealistic, but it's also a waste of our time and resources. In order to streamline the process, I'll demonstrate how to create standard feedforward neural networks using the Keras library.

Finally, we'll round out this chapter with a discussion of the four ingredients you'll need when building *any* neural network.

10.1 Neural Network Basics

Before we can work with Convolutional Neural Networks, we first need to understand the basics of neural networks. In this section we'll review:

- Artificial Neural Networks and their relation to biology.
- The seminal Perceptron algorithm.
- The backpropagation algorithm and how it can be used to train *multi-layer* neural networks efficiently.
- How to train neural networks using the Keras library.

By the time you finish this chapter, you'll have a strong understanding of neural networks and be able to move on to the more advanced Convolutional Neural Networks.

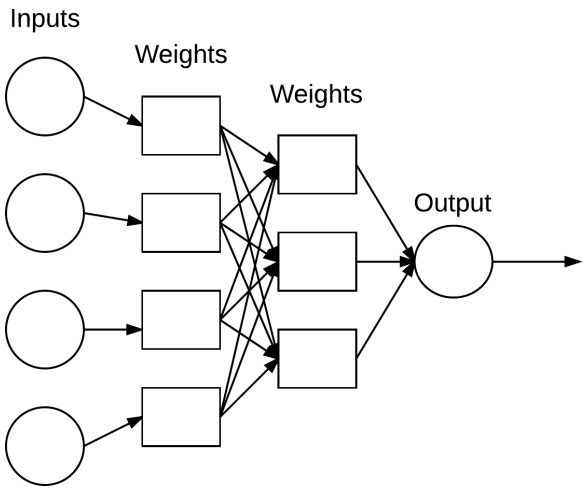


Figure 10.1: A simple neural network architecture. Inputs are presented to the network. Each connection carries a signal through the two hidden layers in the network. A final function computes the output class label.

10.1.1 Introduction to Neural Networks

Neural networks are the building blocks of deep learning systems. In order to be successful at deep learning, we need to start by reviewing the basics of neural networks, including *architecture*, *node types*, and *algorithms for "teaching"* our networks.

In this section we'll start off with a high-level overview of neural networks and the motivation behind them, including their relation to biology in the human mind. From there we'll discuss the most common type of architecture, **feedforward neural networks**. We'll also briefly discuss the concept of *neural learning* and how it will later relate to the algorithms we use to train neural networks.

What are Neural Networks?

Many tasks that involve intelligence, pattern recognition, and object detection are *extremely difficult to automate*, yet *seem to be performed easily and naturally* by animals and young children. For example, how does your family dog recognize you, the owner, versus a complete and total stranger? How does a small child learn to recognize the difference between a *school bus* and a *transit bus*? And how do our own brains subconsciously perform complex pattern recognition tasks each and every day without us even noticing?

The answer lies within our own bodies. Each of us contains a real-life biological neural networks that is connected to our nervous systems – this network is made up of a large number of interconnected *neurons* (nerve cells).

The word *"neural"* is the adjective form of *"neuron"*, and *"network"* denotes a graph-like structure; therefore, an *"Artificial Neural Network"* is a computation system that attempts to mimic (or at least, is inspired by) the neural connections in our nervous system. Artificial neural networks are also referred to as *"neural networks"* or *"artificial neural systems"*. It is common to abbreviate Artificial Neural Network and refer to them as *"ANN"* or simply *"NN"* – I will be using both of the abbreviations throughout the rest of the book.

For a system to be considered an NN, it must contain a labeled, directed graph structure where each node in the graph performs some *simple computation*. From graph theory, we know that a directed graph consists of a set of nodes (i.e., vertices) and a set of connections (i.e., edges) that link together pairs of nodes. In Figure 10.1 we can see an example of such an NN graph.

Each node performs a simple computation. Each connection then carries a *signal* (i.e., the output of the computation) from one node to another, labeled by a *weight* indicating the extent to which the signal is amplified or diminished. Some connections have large, *positive weights* that amplify the signal, indicating that the signal is very important when making a classification. Others have *negative weights,* diminishing the strength of the signal, thus specifying that the output of the node is less important in the final classification. We call such a system an *Artificial Neural Network* if it consists of a graph structure (like in Figure 10.1) with connection weights that are *modifiable* using a learning algorithm.

Relation to Biology

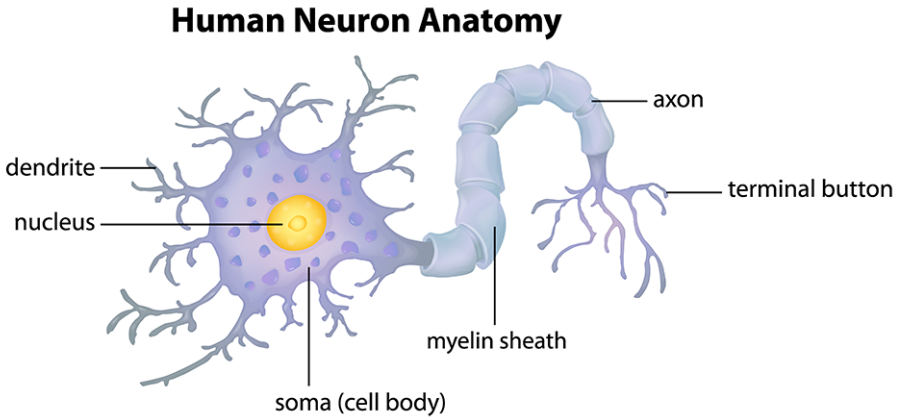


Figure 10.2: The structure of a biological neuron. Neurons are connected to other neurons through their dendrites and enurons.

Our brains are composed of approximately 10 billion neurons, each connected to about 10,000 other neurons. The cell body of the neuron is called the *soma*, where the inputs (*dendrites*) and outputs (*axons*) connect soma to other soma (Figure 10.2).

Each neuron receives electrochemical inputs from other neurons at their dendrites. If these electrical inputs are sufficiently powerful to activate the neuron, then the activated neuron transmits the signal along its axon, passing it along to the dendrites of other neurons. These attached neurons may also fire, thus continuing the process of passing the message along.

The key takeaway here is that a neuron firing is a **binary operation – the neuron either *fires* or it *doesn't fire***. There are no different "grades" of firing. Simply put, a neuron will only fire if the total signal received at the soma exceeds a given threshold.

However, keep in mind that ANNs are simply *inspired* by what we know about the brain and how it works. The goal of deep learning is *not* to mimic how our brains function, but rather take the pieces that we *understand* and allow us to draw similar parallels in our own work. At the end of the day we do not know enough about neuroscience and the deeper functions of the brain to be able to correctly model how the brain works – instead, we take our *inspirations* and move on from there.

Artificial Models

Let's start by taking a look at a basic NN that performs a simple weighted summation of the inputs in Figure 10.3. The values x_1, x_2, and, x_3 are the **inputs** to our NN and typically correspond to a *single row* (i.e., data point) from our design matrix. The constant value 1 is our bias that is assumed to be embedded into the design matrix. We can think of these inputs as the input feature vectors to the NN.

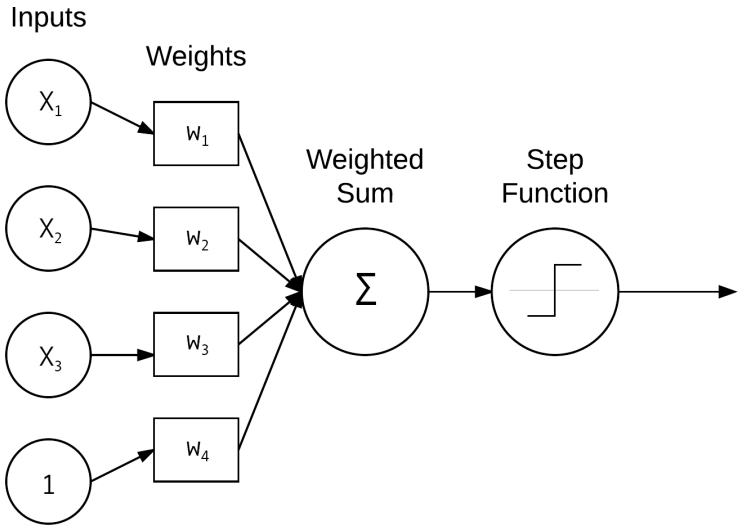


Figure 10.3: A simple NN that takes the weighted sum of the input x and weights w. This weighted sum is then passed through the activation function to determine if the neuron fires.

In practice these inputs could be vectors used to quantify the contents of an image in a systematic, predefined way (e.x., color histograms, Histogram of Oriented Gradients [32], Local Binary Patterns [21], etc.). In the context of deep learning, these inputs are the *raw pixel intensities* of the images themselves.

Each x is connected to a neuron via a weight vector $\boldsymbol{W}$ consists of $w_1, w_2, \ldots w_n$, meaning that for each input x we also have an associated weight w.

Finally, the *output node* on the right of Figure 10.3 takes the weighted sum, applies an activation function f (used to determine if the neuron "fires" or not), and outputs a value. Expressing the output mathematically, you'll typically encounter the following three forms:

- $f(w_1x_1 + w_2x_2 + \cdots + w_nx_n)$
- $f(\sum_{i=1}^{n} w_ix_i)$
- Or simply, $f(net)$, where $net = \sum_{i=1}^{n} w_ix_i$

Regardless how the output value is expressed, understand that we are simply taking the weighted sum of inputs, followed by applying an activation function f.

Activation Functions

The most simple activation function is the "step function", used by the Perceptron algorithm (which we'll cover in the next section).

$$f(net) = \begin{cases} 1 & \text{if } net > 0 \\ 0 & \text{otherwise} \end{cases}$$

As we can see from the equation above, this is a very simple threshold function. If the weighted sum $\sum_{i=1}^{n} w_ix_i > 0$, we output 1, otherwise, we output 0.

Plotting input values along the x-axis and the output of $f(net)$ along the $y-axis$ we can see why this activation function received its name (Figure 10.4, *top-left*). The output of f is always zero when *net* is less than or equal zero. If *net* is greater than to zero, then f will return one. Thus, this function looks like a stair step, not dissimilar to the stairs you walk up and down every day.

However, while being intuitive and easy to use, the step function is not differentiable, which can lead to problems when applying gradient descent and training our network.

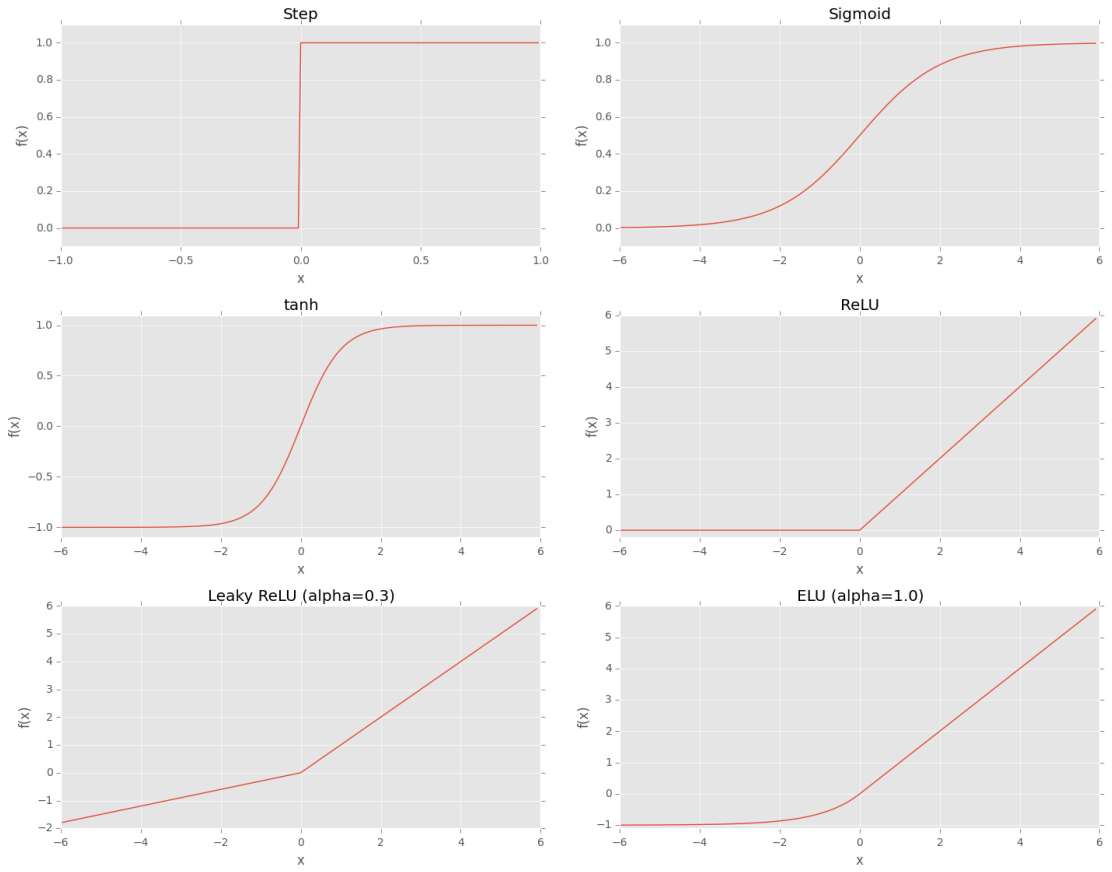


Figure 10.4: **Top-left:** Step function. **Top-right:** Sigmoid activation function. **Mid-left:** Hyperbolic tangent. **Mid-right:** ReLU activation (most used activation function for deep neural networks). **Bottom-left:** Leaky ReLU, variant of the ReLU that allows for negative values. **Bottom-right:** ELU, another variant of ELU that can often perform better than Leaky ReLU.

Instead, a more common activation function used in the history of NN literature is the sigmoid function (Figure 10.4, *top-right*), which follows the equation:

$$t = \sum_{i=1}^{n} w_i x_i \qquad\qquad s(t) = 1/(1+e^{-t}) \tag{10.1}$$

The sigmoid function is a better choice for learning than the simple step function since it:

1. Is continuous and differentiable everywhere.
2. Is symmetric around the y-axis.
3. Asymptotically approaches its saturation values.

The primary advantage here is that the smoothness of the sigmoid function makes it easier to devise learning algorithms. However, there are *two big problems* with the sigmoid function:

1. The outputs of the sigmoid are not zero centered.
2. Saturated neurons essentially kill the gradient, since the delta of the gradient will be extremely small.

The hyperbolic tangent, or *tanh* (with a similar shape of the sigmoid) was also heavily used as an activation function up until the late 1990s (Figure 10.4, *mid-left*): The equation for *tanh* follows:

$$f(z) = tanh(z) = (e^z - e^{-z})/(e^z + e^{-z}) \tag{10.2}$$

The *tanh* function is zero centered, but the gradients are still killed when neurons become saturated.

We now know there are better choices for activation functions than the sigmoid and *tanh* functions. Specifically, the work of Hahnloser et al. in their 2000 paper, *Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit* [101], introduced the **Rectified Linear Unit (ReLU)**, defined as:

$$f(x) = max(0, x) \tag{10.3}$$

ReLU are also called "ramp functions" due to how they look when plotted (Figure 10.4, *mid-right*). Notice how the function is zero for negative inputs but then linearly increases for positive values. The ReLU function is not saturable and is also extremely computationally efficient.

Empirically, the ReLU activation function tends to outperform *both* the sigmoid and *tanh* functions in nearly all applications. Combined with the work of Hahnloser and Seung in their followup 2003 paper *Permitted and Forbidden Sets in Symmetric Threshold-Linear Networks* [102], it was found that the ReLU activation function has stronger biological motivations than the previous families of an activation functions, including more complete mathematical justifications.

As of 2015, ReLU is the *most popular* activation function used in deep learning [9]. However, a problem arises when we have a value of zero – *the gradient cannot be taken*.

A variant of ReLUs, called *Leaky ReLUs* [103] allow for a small, non-zero gradient when the unit is not active:

$$f(net) = \begin{cases} net & \text{if } net >= 0 \\ \alpha \times net & \text{otherwise} \end{cases}$$

Plotting this function in Figure 10.4 (*bottom-left*), we can see that the function is indeed allowed to take on a negative value, unlike traditional ReLUs which "clamp" the function output at zero.

Parametric ReLUs, or PReLUs for short [96], build on Leaky ReLUs and allow the parameter α to be learned on an activation-by-activation basis, implying that each node in the network can learn a different "coefficient of leakage" separate from the other nodes.

Finally, we also have *Exponential Linear Units (ELUs)* introduced by Clevert et al. in their 2015 paper, *Fast and Accurate Deep Learning by Exponential Linear Units (ELUs)* [104]:

$$f(net) = \begin{cases} net & \text{if } net >= 0 \\ \alpha \times (exp(net) - 1) & \text{otherwise} \end{cases}$$

The value of α is constant and *set when the network architecture is instantiated* – this is unlike PReLUs where α is learned. A typical value for α is $\alpha = 1.0$. Figure 10.4 (*bottom-right*) visualizes the ELU activation function.

Through the work of Clevert et al. (and my own anecdotal experiments), ELUs often obtain higher classification accuracy than ReLUs. ELUs rarely, if ever perform worse than your standard ReLU function.

Which Activation Function Do I Use?

Given the popularity of the most recent incarnation of deep learning, there has been an associated explosion in activation functions. Due to the number of choices of activation functions, both modern (ReLU, Leaky ReLU, ELU, etc.) and "classical" ones (step, sigmoid, *tanh*, etc.), it may appear to be a daunting, perhaps even overwhelming task to select an appropriate activation function.

However, in nearly all situations, I recommend starting with a ReLU to obtain a baseline accuracy (as do most papers published in the deep learning literature). From there you can try swapping out your standard ReLU for a Leaky ReLU variant.

My personal preference is to start with a ReLU, tune my network and optimizer parameters (architecture, learning rate, regularization strength, etc.) and note the accuracy. Once I am reasonably satisfied with the accuracy, I swap in an ELU and often notice a $1-5\%$ improvement in classification accuracy depending on the dataset. Again, this is only my anecdotal advice. You should run your own experiments and note your findings, but as a general rule of thumb, start with a normal ReLU and tune the other parameters in your network – then swap in some of the more "exotic" ReLU variants.

Feedforward Network Architectures

While there are many, *many* different NN architectures, the most common architecture is the *feedforward network*, as presented in Figure 10.5.

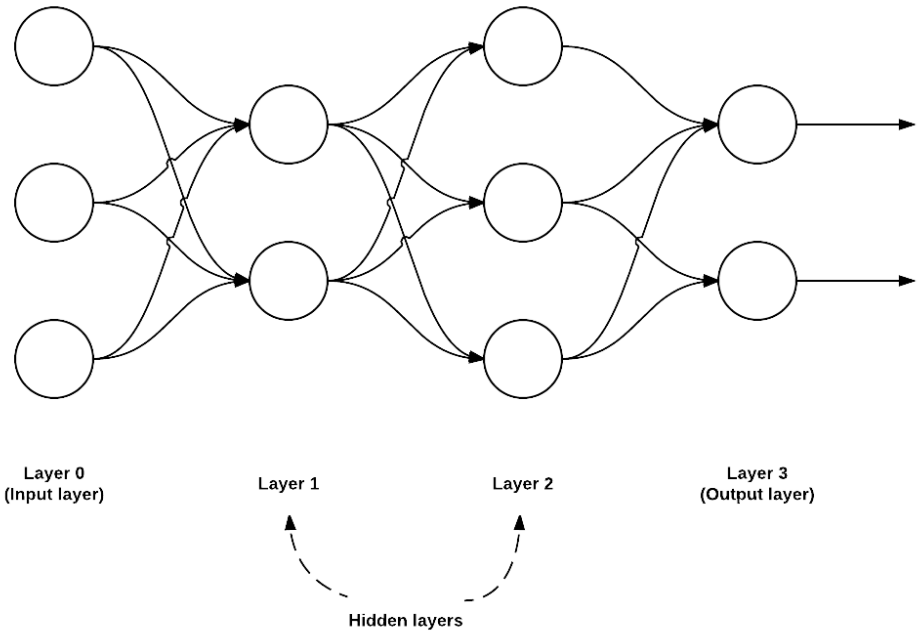


Figure 10.5: An example of a feedforward neural network with 3 input nodes, a hidden layer with 2 nodes a second hidden layer with 3 nodes, and a final output layer with 2 nodes.

In this type of architecture, a connection between nodes is *only allowed* from nodes in layer i to nodes in layer $i+1$ (hence the term, *feedforward*). There are no backward or inter-layer connections allowed. When feedforward networks include *feedback connections* (output connections that feed back into the inputs) they are called **recurrent neural networks**.

In this book we focus on feedforward neural networks as they are the cornerstone of modern deep learning applied to computer vision. As we'll find out in Chapter 11, Convolutional Neural Networks are simply a special case of feedforward neural network.

To describe a feedforward network, we normally use a sequence of integers to quickly and concisely depot the number of nodes in each layer. For example, the network in Figure 10.5 above is a *3-2-3-2* feedforward network:

Layer 0 contains 3 inputs, our x_i values. These could be raw pixel intensities of an image or a feature vector extracted from the image.

Layers 1 and 2 are *hidden layers* containing 2 and 3 nodes, respectively.

Layer 3 is the *output layer or the visible layer* – there is where we obtain the overall output classification from our network. The output layer typically has as many nodes as class labels; one node for each potential output. For example, if we were to build an NN to classify handwritten digits, our output layer would consist of 10 nodes, one for each digit *0-9*.

Neural Learning

Neural learning refers to the method of modifying the weights and connections between nodes in a network. Biologically, we define learning in terms of Hebb's principle:

> *"When an axon of cell A is near enough to excite cell B, and repeatedly or persistently takes place in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased"* – Donald Hebb [105]

In terms of ANNs, this principle implies that there should be an increase in strength of connections between nodes that have similar outputs when presented with the same input. We call this *correlation learning* because the strength of the connections between neurons eventually represents the correlation between outputs.

What are Neural Networks Used For?

Neural Networks can be used in both supervised, unsupervised, and semi-supervised learning tasks, providing the appropriate architecture is used, of course. A complete review of NNs is outside the scope of this book (please see Schmidhuber [40] for an extensive survey of deep artificial networks, along with Mehrota [106] for an review of classical methods); however, common applications of NN include classification, regression, clustering, vector quantization, pattern association, and function approximation, just to name a few.

In fact, for nearly every facet of machine learning, NNs have been applied in some form or another. In the context of this book, we'll be using NNs for *computer vision* and *image classification*.

A Summary of Neural Network Basics

In this section, we reviewed the basics of Artificial Neural Networks (ANNs, or simply NNs). We started by examining the biological motivation behind ANNs, and then learned how we can *mathematically define* a function to mimic the activation of a neuron (i.e., the activation function).

Based on this model of a neuron, we are able to define the *architecture* of a network consisting of (at a bare minimum), an *input layer* and an *output layer*. Some network architectures may include *multiple hidden layers* between the input and output layers. Finally, each layer can have one or more nodes. Nodes in the input layer *do not* contain an activation function (they are "where" the individual pixel intensities of our image are inputted); however, nodes in both the hidden and output layers *do* contain an activation function.

We also reviewed three popular activation functions: *sigmoid*, *tanh*, and *ReLU* (and its variants).

Traditionally the sigmoid and *tanh* functions have been used to train networks; however, since Hahnloser et al.'s 2000 paper [101], the ReLU function has been used more often.

In 2015, ReLU is *by far* the most popular activation function used in deep learning architectures [9]. Based on the success of ReLU, we also have *Leaky ReLUs*, a variant of ReLUs that seek to

improve network performance by allowing the function to take on a negative value. The Leaky ReLU family of functions consists of your standard leaky ReLU variant, PReLUs, and ELUs.

Finally, it's important to note that even though we are focusing on deep learning strictly in the context of *image classification*, neural networks have been used in some fashion in nearly all niches of machine learning.

Now that we understand the basics of NNs, let's make this knowledge more concrete by examining actual architectures and their associated implementations. In the next section, we'll discuss the classic Perceptron algorithm, one of the first ANNs ever to be created.

10.1.2 The Perceptron Algorithm

First introduced by Rosenblatt in 1958, *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain* [12] is arguably the oldest and most simple of the ANN algorithms. Following this publication, Perceptron-based techniques were all the rage in the neural network community. This paper alone is hugely responsible for the popularity and utility of neural networks today.

But then, in 1969, an "AI Winter" descended on the machine learning community that almost froze out neural networks for good. Minsky and Papert published *Perceptrons: an introduction to computational geometry* [14], a book that effectively stagnated research in neural networks for almost a decade – there is much controversy regarding the book [107], but the authors did successfully demonstrate that a *single layer* Perceptron is unable to separate nonlinear data points.

Given that most real-world datasets are naturally nonlinearly separable, this it seemed that the Perceptron, along with the rest of neural network research, might reach an untimely end.

Between the Minsky and Papert publication and the broken promises of neural networks revolutionizing industry, the interest in neural networks dwindled substantially. It wasn't until we started exploring deeper networks (sometimes called *multi-layer perceptrons*) along with the backpropagation algorithm (Werbos [15] and Rumelhart [16]) that the "AI Winter" in the 1970s ended and neural network research started to heat up again.

All that said, the Perceptron is still a *very important algorithm* to understand as it sets the stage for more advanced multi-layer networks. We'll start this section with a review of the Perceptron architecture and explain the training produced (called the **delta rule**) used to train the Perceptron. We'll also look at the *termination criteria* of the network (i.e., when the Perceptron should stop training). Finally, we'll implement the Perceptron algorithm in pure Python and use it to study and examine how the network is unable to learn nonlinearly separable datasets.

AND, OR, and XOR Datasets

Before we study the Perceptron itself, let's first discuss "bitwise operations", including AND, OR, and XOR (exclusive OR). If you've taken an introductory level computer science course before you might already be familiar with bitwise functions.

Bitwise operators and associated bitwise datasets accept two input bits and produce a final output bit after applying the operation. Given two input bits, each potentially taking on a value of `0` or `1`, there are four possible combinations of of these two bits – Table 10.1 provides the possible input and output values for AND, OR, and XOR:

As we can see on the *left*, a logical AND is true *if and only if* both input values are `1`. If *either* of the input values are `0`, the AND returns `0`. Thus, there is only one combination, $x_0 = 1$ and $x_1 = 1$ when the output of AND is true.

In the *middle*, we have the OR operation which is true when only *one* of the input values is `1`. Thus, there are three possible combinations of the two bits x_0 and x_1 that produce a value of $y = 1$.

Finally, the *right* displays the XOR operation which is true *if and only if* one if the inputs is `1` *but not both*. While OR had three possible situations where $y = 1$, XOR only has two.

x_0	x_1	$x_0 \& x_1$
0	0	0
0	1	0
1	0	0
1	1	1

x_0	x_1	$x_0 \mid x_1$
0	0	0
0	1	1
1	0	1
1	1	1

x_0	x_1	$x_0 \wedge x_1$
0	0	0
0	1	1
1	0	1
1	1	0

Table 10.1: **Left:** The bitwise AND dataset. Given two inputs, the output is only 1 if both inputs are 1. **Middle:** The bitwise OR dataset. Given two inputs, the output is 1 if *either* of the two inputs is 1. **Right:** The XOR (e(X)clusive OR) dataset. Given two inputs, the output 1 if and only if one if the inputs is 1, *but not both*.

We often use these simple "bitwise datasets" to test and debug machine learning algorithms. If we plot and visualize the AND, OR, and XOR values (with red circles being zero outputs and blue stars one outputs) in Figure 10.6, you'll notice an interesting pattern:

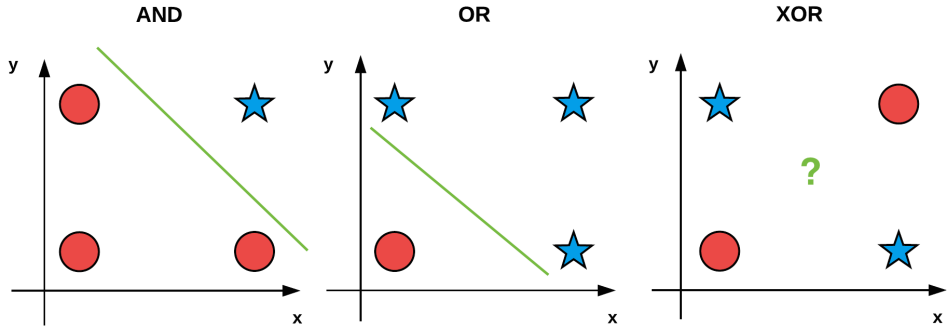


Figure 10.6: Both the AND and OR bitwise datasets are linearly separable, meaning that we can draw a *single line* (green) that separates the two classes. However, for XOR it is impossible to draw a single line that separates the two classes – this is therefore a nonlinearly separable dataset.

Both AND and OR are linearly separable – we can clearly draw a line that separates the `0` and `1` classes – the same is not true for XOR. Take the time now to convince yourself that it is *not possible* to draw a line that cleanly separates the two classes in the XOR problem. XOR is, therefore, an example of a *nonlinearly separable* dataset.

Ideally, we would like our machine learning algorithms to be able to separate nonlinear classes as most datasets encountered in the real-world are nonlinear. Therefore, when constructing, debugging, and evaluating a given machine learning algorithm, we may use the bitwise values x_0 and x_1 as our *design matrix* and then try to predict the corresponding y values.

Unlike our standard procedure of splitting our data into *training* and *testing* splits, when using bitwise datasets we simply train and evaluate our network on the same set of data. Our goal here is simply to determine if it's even *possible* for our learning algorithm to learn the patterns in the data. As we'll find out, the Perceptron algorithm can correctly classify the AND and OR functions but fails to classify the XOR data.

Perceptron Architecture

Rosenblatt [12] defined a Perceptron as a system that learns using labeled examples (i.e., supervised learning) of feature vectors (or raw pixel intensities), mapping these inputs to their corresponding output class labels.

In its simplest form, a Perceptron contains N input nodes, one for each entry in the *input row* of

the design matrix, followed by *only one layer* in the network with just a *single node* in that layer (Figure 10.7).

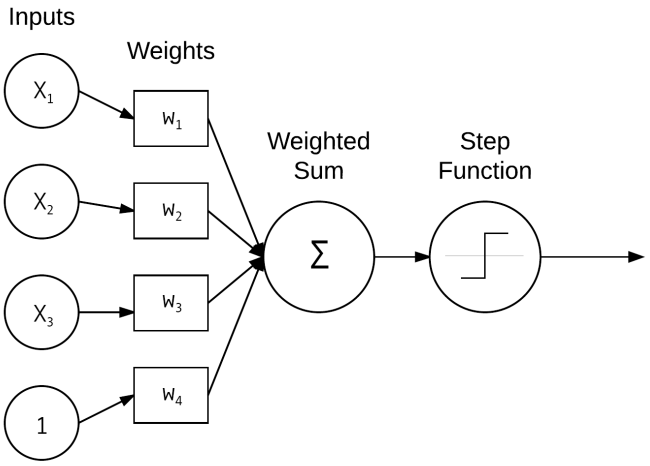


Figure 10.7: Architecture of the Perceptron network.

There exist connections and their corresponding weights $w_1, w_2, \ldots w_i$ from the input x_i's to the single output node in the network. This node takes the weighted sum of inputs and applies a *step function* to determine the output class label. The Perceptron outputs either a `0` or a `1` – `0` for class #1 and `1` for class #2; thus, in its original form, the Perceptron is simply a binary, two class classifier.

Perceptron Training Procedure and the Delta Rule

Training a Perceptron is a fairly straightforward operation. Our goal is to obtain a set of weights $\boldsymbol{w}$ that accurately classifies each instance in our training set. In order to train our Perceptron, we iteratively feed the network our training data *multiple times*. Each time the network has seen the *full set* of training data, we say an *epoch* has passed. It normally takes many epochs until a weight vector $\boldsymbol{w}$ can be learned to linearly separate our two classes of data.

The pseudocode for the Perceptron training algorithm can be found below:

The actual "learning" takes place in Steps 2b and 2c. First, we pass the feature vector x_j through the network, take the dot product weight the weights $\boldsymbol{w}$ and obtain the output y_j. This value is then passed through the step function which will return 1 if $x > 0$ and 0 otherwise.

Now we need to update our weight vector $\boldsymbol{w}$ to step in the direction that is "closer" to the correct classification. This update of the weight vector is handled by the *delta rule* in Step 2c.

The expression $(d_j - y_j)$ determines if the output classification is correct or not. If the classification is *correct*, then this difference will be zero. Otherwise, the difference will be either positive or negative, giving us the direction in which our weights will be updated (ultimately bringing us closer to the correct classification). We then multiply $(d_j - y_j)$ by x_j, moving us closer to the correct classification.

1. Initialize our weight vector $\boldsymbol{w}$ with small random values
2. Until Perceptron converges:
   (a) Loop over each feature vector x_j and true class label d_i in our training set D
   (b) Take $\boldsymbol{x}$ and pass it through the network, calculating the output value: $y_j = f(\boldsymbol{w(t)} \cdot \boldsymbol{x}_j)$
   (c) Update the weights $\boldsymbol{w}$: $w_i(t+1) = w_i(t) + \eta(d_j - y_j)x_{j,i}$ for all features $0 <= i <= n$

Figure 10.8: The Perceptron algorithm training procedure.

The value α is our *learning rate* and controls how large (or small) of a step we take. It's *critical* that this value is set correctly. A larger value of α will cause us to take a step in the right direction; however, this step could be *too large*, and we could easily overstep a local/global optimum.

Conversely, a small value of α allows us to take tiny baby steps in the right direction, ensuring we don't overstep a local/global minimum; however, these tiny baby steps make take an intractable amount of time for our learning to converge.

Finally, we add in the previous weight vector at time t, $\mathbf{w}_{j(t)}$ which completes the process of "stepping" towards the correct classification. If you find this training procedure a bit confusing, don't worry – we'll be covering it in detail with Python code later in Section 10.1.2.

Perceptron Training Termination

The Perceptron training process is allowed to proceed until all training samples are classified correctly *or* a preset number of epochs is reached. Termination is ensured if α is sufficiently small *and* the training data is linearly separable.

So, what happens if our data is not linearly separable or we make a poor choice in α? Will training continue infinitely? In this case, no – we normally stop after a set number of epochs has been hit or if the number of misclassifications has not changed in a large number of epochs (indicating that the data is not linearly separable). For more details on the perceptron algorithm, please refer to either Andrew Ng's Stanford lecture [76] or the introductory chapters of Mehrota et al. [106].

Implementing the Perceptron in Python

Now that we have studied the Perceptron algorithm, let's implement the actual algorithm in Python. Create a file named `perceptron.py` in your `pyimagesearch.nn` package – this file will store our actual `Perceptron` implementation:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- nn
|    |    |--- __init__.py
|    |    |--- perceptron.py
```

After you've created the file, open it up, and insert the following code:

```
1  # import the necessary packages
2  import numpy as np
3
4  class Perceptron:
5      def __init__(self, N, alpha=0.1):
6          # initialize the weight matrix and store the learning rate
7          self.W = np.random.randn(N + 1) / np.sqrt(N)
8          self.alpha = alpha
```

Line 5 defines the constructor to our `Perceptron` class, which accepts a single required parameter followed by a second optional one:

1. `N`: The number of columns in our input feature vectors. In the context of our bitwise datasets, we'll set `N` equal to two since there are two inputs.
2. `alpha`: Our learning rate for the Perceptron algorithm. We'll set this value to `0.01` by default. Common choices of learning rates are normally in the range $\alpha = 0.1, 0.01, 0.001$.

Line 7 files our weight matrix `W` with random values sampled from a "normal" (Gaussian) distribution with zero mean and unit variance. The weight matrix will have $N + 1$ entries, one for each of the `N` inputs in the feature vector, plus one for the bias. We divide `W` by the square-root of the number of inputs, a common technique used to scale our weight matrix, leading to faster convergence. We will cover weight initialization techniques later in this chapter.

Next, let's define the `step` function:

```
10     def step(self, x):
11         # apply the step function
12         return 1 if x > 0 else 0
```

This function mimics the behavior of the step equation in Section 10.4 above – if `x` is positive we return `1`, otherwise we return `0`.

To actually train the Perceptron we'll define a function named `fit`. If you have any previous experience with machine learning, Python, and the scikit-learn library then you'll know that it's common to name your training procedure function `fit`, as in *"fit a model to the data"*:

```
14     def fit(self, X, y, epochs=10):
15         # insert a column of 1's as the last entry in the feature
16         # matrix -- this little trick allows us to treat the bias
17         # as a trainable parameter within the weight matrix
18         X = np.c_[X, np.ones((X.shape[0]))]
```

The `fit` method requires two parameters followed by a single optional one:

The `X` value is our actual training data. The `y` variable is our target output class labels (i.e., what our network *should* be predicting). Finally, we supply `epochs`, the number of epochs our Perceptron will train for.

Line 18 applies the bias trick (Section 9.3) by inserting a column of ones into the training data, which allows us to treat the bias as a trainable parameter *directly* inside the weight matrix.

Next, let's review the actual training procedure:

```
20         # loop over the desired number of epochs
21         for epoch in np.arange(0, epochs):
22             # loop over each individual data point
23             for (x, target) in zip(X, y):
24                 # take the dot product between the input features
25                 # and the weight matrix, then pass this value
26                 # through the step function to obtain the prediction
27                 p = self.step(np.dot(x, self.W))
28
29                 # only perform a weight update if our prediction
30                 # does not match the target
31                 if p != target:
32                     # determine the error
33                     error = p - target
34
35                     # update the weight matrix
36                     self.W += -self.alpha * error * x
```

On **Line 21** we start looping over the desired number of `epochs`. For each epoch, we also loop over each individual data point `x` and output `target` class label (**Line 23**).

Line 27 takes the dot product between the input features `x` and the weight matrix `W`, then passes the output through the `step` function to obtain the prediction by the Perceptron.

Applying the same training procedure detailed in Listing 10.8 above, we only perform a weight update if our prediction *does not* match the target (**Line 31**). If this is the case, we determine the `error` (**Line 33**) by computing the sign (either positive or negative) via the difference operation.

Updating the weight matrix is handled on **Line 36** where we take a step towards the correct classification, scaling this step by our learning rate `alpha`. Over a series of epochs, our Perceptron is able to learn patterns in the underlying data and shift the values of the weight matrix such that we correctly classify our input samples `x`.

The last function we need to define is `predict`, which, as the name suggests, is used to *predict* the class labels for a given set of input data:

```
38      def predict(self, X, addBias=True):
39          # ensure our input is a matrix
40          X = np.atleast_2d(X)
41
42          # check to see if the bias column should be added
43          if addBias:
44              # insert a column of 1's as the last entry in the feature
45              # matrix (bias)
46              X = np.c_[X, np.ones((X.shape[0]))]
47
48          # take the dot product between the input features and the
49          # weight matrix, then pass the value through the step
50          # function
51          return self.step(np.dot(X, self.W))
```

Our `predict` method requires a set of input data `X` that needs to be classified. A check on **Line 43** is made to see if a bias column needs to be added.

Obtaining the output predictions for `X` is the same as the training procedure – simply take the dot product between the input features `X` and our weight matrix `W`, and then pass the value through our step function. The output of the step function is returned to the calling function.

Now that we've implemented our `Perceptron` class, let's try to apply it to our bitwise datasets and see how the neural network performs.

Evaluating the Perceptron Bitwise Datasets

To start, let's create a file named `perceptron_or.py` that attempts to fit a Perceptron model to the bitwise OR dataset:

```
1  # import the necessary packages
2  from pyimagesearch.nn import Perceptron
3  import numpy as np
4
5  # construct the OR dataset
6  X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
7  y = np.array([[0], [1], [1], [1]])
8
9  # define our perceptron and train it
10 print("[INFO] training perceptron...")
11 p = Perceptron(X.shape[1], alpha=0.1)
12 p.fit(X, y, epochs=20)
```

Lines 2 and 3 import our required Python packages. We'll be using our `Perceptron` implementation from Section 10.1.2 above. **Lines 6 and 7** define the OR dataset based on Table 10.1.

Lines 11 and 12 train our Perceptron with a learning rate of $\alpha = 0.1$ for a total of 20 epochs.

We can then evaluate our Perceptron on the data to validate that it did, in fact, learn the OR function:

```
14 # now that our perceptron is trained we can evaluate it
15 print("[INFO] testing perceptron...")
16
17 # now that our network is trained, loop over the data points
18 for (x, target) in zip(X, y):
19       # make a prediction on the data point and display the result
20       # to our console
21       pred = p.predict(x)
22       print("[INFO] data={}, ground-truth={}, pred={}".format(
23             x, target[0], pred))
```

On **Line 18** we loop over each of the data points in the OR dataset. For each of these data points, we pass it through the network and obtain the prediction (**Line 21**).

Finally, **Lines 22 and 23** display the input data point, the ground-truth label, as well as our predicted label to our console.

To see if our Perceptron algorithm is able to learn the OR function, just execute the following command:

```
$ python perceptron_or.py
[INFO] training perceptron...
[INFO] testing perceptron...
[INFO] data=[0 0], ground-truth=0, pred=0
[INFO] data=[0 1], ground-truth=1, pred=1
[INFO] data=[1 0], ground-truth=1, pred=1
[INFO] data=[1 1], ground-truth=1, pred=1
```

Sure enough, our neural network is able to correctly predict that the OR operation for $x_0 = 0$ and $x_1 = 0$ is zero – all other combinations are one.

Now, let's move on to the AND function – create a new file named `perceptron_and.py` and insert the following code:

```
1  # import the necessary packages
2  from pyimagesearch.nn import Perceptron
3  import numpy as np
4
5  # construct the AND dataset
6  X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
7  y = np.array([[0], [0], [0], [1]])
8
9  # define our perceptron and train it
10 print("[INFO] training perceptron...")
11 p = Perceptron(X.shape[1], alpha=0.1)
12 p.fit(X, y, epochs=20)
13
```

```
14 # now that our perceptron is trained we can evaluate it
15 print("[INFO] testing perceptron...")
16
17 # now that our network is trained, loop over the data points
18 for (x, target) in zip(X, y):
19       # make a prediction on the data point and display the result
20       # to our console
21       pred = p.predict(x)
22       print("[INFO] data={}, ground-truth={}, pred={}".format(
23             x, target[0], pred))
```

Notice here that the *only* lines of code that have changed are **Lines 6 and 7** where we define the AND dataset rather than the OR dataset.

Executing the following command, we can evaluate the Perceptron on the AND function:

```
$ python perceptron_and.py
[INFO] training perceptron...
[INFO] testing perceptron...
[INFO] data=[0 0], ground-truth=0, pred=0
[INFO] data=[0 1], ground-truth=0, pred=0
[INFO] data=[1 0], ground-truth=0, pred=0
[INFO] data=[1 1], ground-truth=1, pred=1
```

Again, our Perceptron was able to correctly model the function. The AND function is only true when *both* $x_0 = 1$ and $x_1 = 1$ – for all other combinations the bitwise AND is zero.

Finally, let's take a look at the nonlineary separable XOR function inside `perceptron_xor.py`:

```
1  # import the necessary packages
2  from pyimagesearch.nn import Perceptron
3  import numpy as np
4
5  # construct the XOR dataset
6  X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
7  y = np.array([[0], [1], [1], [0]])
8
9  # define our perceptron and train it
10 print("[INFO] training perceptron...")
11 p = Perceptron(X.shape[1], alpha=0.1)
12 p.fit(X, y, epochs=20)
13
14 # now that our perceptron is trained we can evaluate it
15 print("[INFO] testing perceptron...")
16
17 # now that our network is trained, loop over the data points
18 for (x, target) in zip(X, y):
19       # make a prediction on the data point and display the result
20       # to our console
21       pred = p.predict(x)
22       print("[INFO] data={}, ground-truth={}, pred={}".format(
23             x, target[0], pred))
```

Again, the only lines of code that have been changed are **Lines 6 and 7** where we define the XOR data. The XOR operator is true *if and only if* one (but not both) x's are one.

Executing the following command we can see that the Perceptron *cannot* learn this nonlinear relationship:

```
$ python perceptron_xor.py
[INFO] training perceptron...
[INFO] testing perceptron...
[INFO] data=[0 0], ground-truth=0, pred=1
[INFO] data=[0 1], ground-truth=1, pred=1
[INFO] data=[1 0], ground-truth=1, pred=0
[INFO] data=[1 1], ground-truth=0, pred=0
```

No matter how many times you run this experiment with varying learning rates or different weight initialization schemes, you will *never* be able to correctly model the XOR function with a single layer Perceptron. Instead, what we need is *more layers* – and with that, comes the start of deep learning.

10.1.3 Backpropagation and Multi-layer Networks

Backpropagation is arguably *the* most important algorithm in neural network history – without (efficient) backpropagation, it would be *impossible* to train deep learning networks to the depths that we see today. Backpropagation can be considered the cornerstone of modern neural networks and deep learning.

The original incarnation of backpropagation was introduced back in the 1970s, but it wasn't until the seminal 1986 paper, *Learning representations by back-propagating errors* by Rumelhart, Hinton, and Williams [16], were we able to devise a faster algorithm, more adept to training deeper networks.

There are quite literally hundreds (if not thousands) of tutorials on backpropagation available today. Some of my favorites include:

1. Andrew Ng's discussion on backpropagation inside the Machine Learning course by Coursera [76].
2. The heavily mathematically motivated Chapter 2 – *How the backpropagation algorithm works* from *Neural Networks and Deep Learning* by Michael Nielsen [108].
3. Stanford's cs231n exploration and analysis of backpropagation [57].
4. Matt Mazur's excellent concrete example (with actual worked numbers) that demonstrate how backpropagation works [109].

As you can see, there are no shortage of backpropagation guides – instead of regurgitating and reiterating what has been said by others hundreds of times before, I'm going to take a different approach and do what makes PyImageSearch publications special:

Construct an intuitive, easy to follow implementation of the backpropagation algorithm using the Python language.

Inside this implementation, we'll build an actual neural network and train it using the backpropagation algorithm. By the time you finish this section, you'll understand how backpropagation works – and perhaps more importantly, you'll have a stronger understanding of how this algorithm is used to train neural networks from scratch.

Backpropagation

The backpropagation algorithm consists of two phases:

1. The *forward pass* where our inputs are passed through the network and output predictions obtained (also known as the *propagation* phase).

x_0	x_1	y
0	0	0
0	1	1
1	0	1
1	1	0

x_0	x_1	x_2
0	0	1
0	1	1
1	0	1
1	1	1

Table 10.2: **Left:** The bitwise XOR dataset (including class labels). **Right:** The XOR dataset design matrix with a bias column inserted (excluding class labels for brevity).

2. The *backward pass* where we compute the gradient of the loss function at the final layer (i.e., predictions layer) of the network and use this gradient to recursively apply the chain rule to update the weights in our network (also known as the *weight update* phase).

We'll start by reviewing each of these phases at a high level. From there, we'll implement the backpropagation algorithm using Python. Once we have implemented backpropagation we'll want to be able to make *predictions* using our network – this is simply the forward pass phase, only with a small adjustment (in terms of code) to make the predictions more efficient.

Finally, I'll demonstrate how to train a custom neural network using backpropagation and Python on both the:

1. XOR dataset
2. MNIST dataset

The Forward Pass

The purpose of the forward pass is to propagate our inputs through the network by applying a series of dot products and activations until we reach the output layer of the network (i.e., our predictions). To visualize this process, let's first consider the XOR dataset (Table 10.2, *left*).

Here we can see that each entry X in the design matrix (left) is 2-dim – each data point is represented by *two* numbers. For example, the first data point is represented by the feature vector $(0,0)$, the second data point by $(0,1)$, etc. We then have our output values y as the right column. Our target output values are the *class labels*. Given an input from the design matrix, our goal is to correctly predict the target output value.

As we'll find out in Section 10.1.3 below, to obtain perfect classification accuracy on this problem we'll need a feedforward neural network with at least a single hidden layer, so let's go ahead and start with a $2-2-1$ architecture (Figure 10.9, *top*). This is a good start; however, we're forgetting to include the bias term. As we know from Chapter 9, there are two ways to include the bias term $\boldsymbol{b}$ in our network. We can either:

1. Use a separate variable.
2. Treat the bias as a trainable parameter *within* the weight matrix by inserting a column of 1's into the feature vectors.

Inserting a column of 1's into our feature vector is done programmatically, but to ensure we understand this point, let's update our XOR design matrix to explicitly see this taking place (Table 10.2, *right*). As you can see, a column of 1's have been added to our feature vectors. In practice you can insert this column anywhere you like, but we typically place it either as (1) the first entry in the feature vector or (2) the last entry in the feature vector.

Since we have changed the size of our input feature vector (normally performed *inside* neural network implementation itself so that we do not need to explicitly modify our design matrix), that changes our (perceived) network architecture from $2-2-1$ to an (internal) $3-3-1$ (Figure 10.9, *bottom*).

We'll still refer to this network architecture as $2-2-1$, but when it comes to implementation, it's actually $3-3-1$ due to the addition of the bias term embedded in the weight matrix.

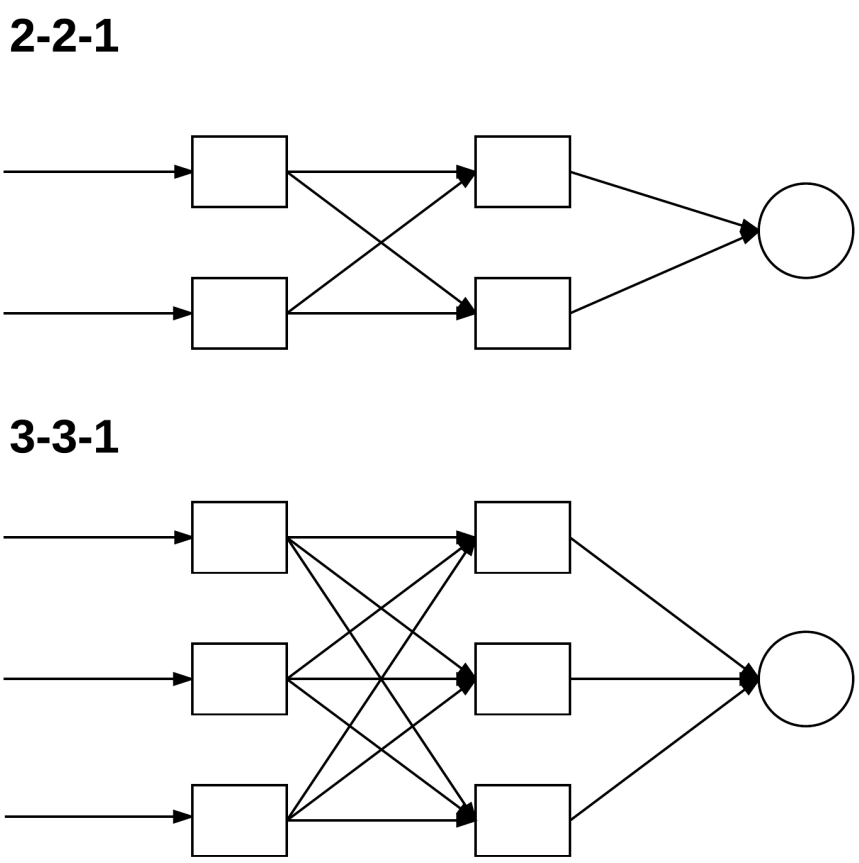


Figure 10.9: **Top:** To build a neural network to correctly classify the XOR dataset, we'll need a network with two input nodes, two hidden nodes, and one output node. This gives rise to a $2-2-1$ architecture. **Bottom:** Our actual internal network architecture representation is $3-3-1$ due to the bias trick. In the vast majority of neural network implementations this adjustment to the weight matrix happens internally and is something that you do not need to worry about; however, it's still important to understand what is going on under the hood.

Finally, recall that both our input layer and all hidden layers require a bias term; however, the final output layer *does not* require a bias. The benefit of applying the bias trick is that we do not need to explicitly keep track of the bias parameter any longer – it is now a trainable parameter *within* the weight matrix, thus making training more efficient and substantially easier to implement. Please see Chapter 9 for a more thorough discussion on why this bias trick works.

To see the forward pass in action, we first initialize the weights in our network, as in Figure 10.10. Notice how each arrow in the weight matrix has a value associated with it – this is the *current* weight value for a given node and signifies the amount in which a given input is amplified or diminished. This weight value will then be *updated* during the backpropagation phase.

On the far left of Figure 10.10, we present the feature vector $(0, 1, 1)$ (and target output value 1 to the network). Here we can see that 0, 1, and 1 have been assigned to the three input nodes in the network. To propagate the values through the network and obtain the final classification, we need to take the dot product between the inputs and the weight values, followed by applying an activation function (in this case, the *sigmoid* function, σ).

Let's compute the inputs to the three nodes in the hidden layers:

1. $\sigma((0 \times 0.351) + (1 \times 1.076) + (1 \times 1.116)) = 0.899$

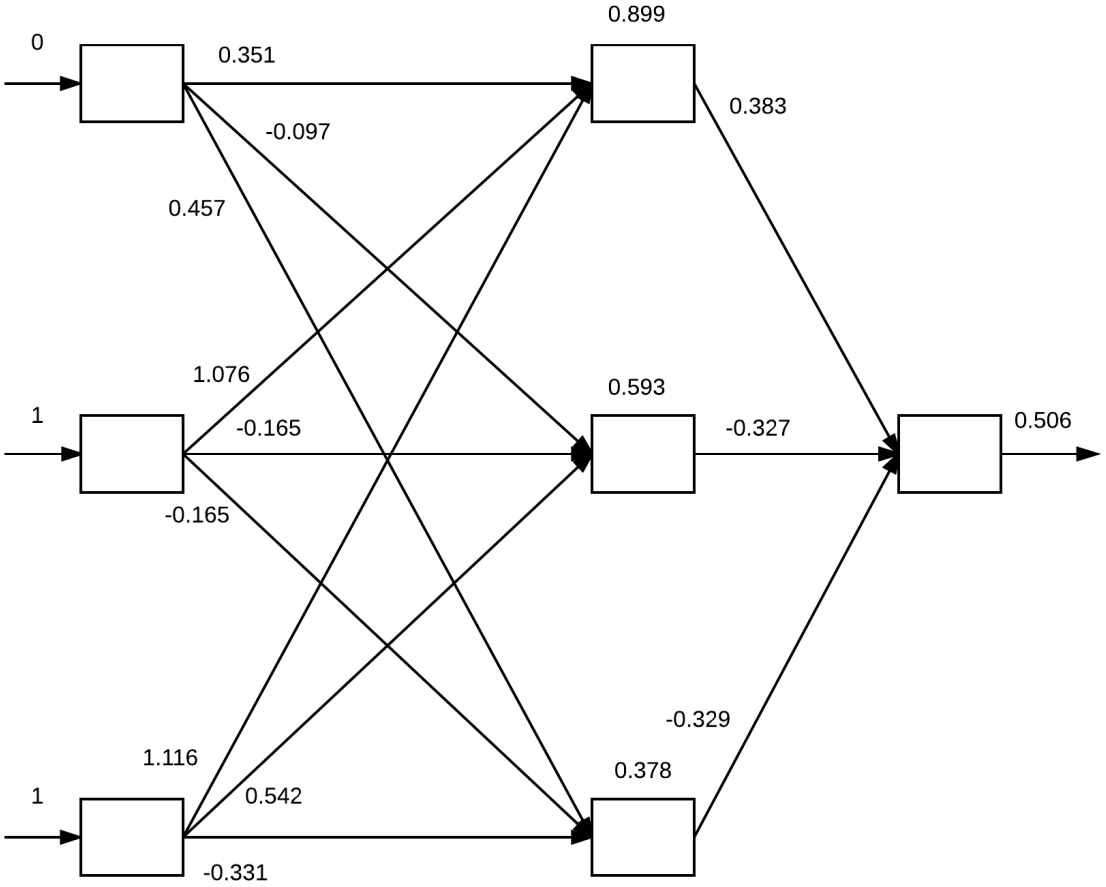


Figure 10.10: An example of the forward propagation pass. The input vector $[0,1,1]$ is presented to the network. The dot product between the inputs and weights are taken, followed by applying the sigmoid activation function to obtain the values in the hidden layer (0.899, 0.593, and 0.378, respectively). Finally, the dot product and sigmoid activation function is computed for the final layer, yielding an output of 0.506. Applying the step function to 0.506 yields 1, which is indeed the correct target class label.

2. $\sigma((0 \times -0.097) + (1 \times -0.165) + (1 \times 0.542)) = 0.593$
3. $\sigma((0 \times 0.457) + (1 \times -0.165) + (1 \times -0.331)) = 0.378$

Looking at the node values of the hidden layers (Figure 10.10, *middle*), we can see the nodes have been updated to reflect our computation.

We now have our *inputs* to the hidden layer nodes. To compute the output prediction, we once again compute the dot product followed by a sigmoid activation:

$$\sigma((0.899 \times 0.383) + (0.593 \times -0.327) + (0.378 \times -0.329)) = 0.506 \tag{10.4}$$

The output of the network is thus 0.506. We can apply a step function to determine if this output is the correct classification or not:

$$f(net) = \begin{cases} 1 & \text{if } net > 0 \\ 0 & \text{otherwise} \end{cases}$$

Applying the step function with $net = 0.506$ we see that our network predicts 1 which is, in fact, the correct class label. However, our network is *not very confident* in this class label – the predicted value 0.506 is very close to the threshold of the step. Ideally, this prediction should be closer to $0.98 - 0.99$., implying that our network has truly learned the underlying pattern in the dataset. In order for our network to actually "learn", we need to apply the backward pass.

The Backward Pass

In order to apply the backpropagation algorithm, our activation function must be *differentiable* so that we can compute the *partial derivative* of the error with respect to a given weight $w_{i,j}$, loss (E), node output o_j, and network output net_j.

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_{i,j}} \tag{10.5}$$

As the calculus behind backpropagation has been exhaustively explained many times in previous works (see Andrew Ng [76], Michael Nielsen [108], Matt Mazur [109]), I'm going to skip the derivation of the backpropagation chain rule update and instead explain it via code in the following section.

For the mathematically astute, please see the references above for more information on the chain rule and its role in the backpropagation algorithm. By explaining this process in code, my goal is to help readers understand backpropagation through a more intuitive, implementation sense.

Implementing Backpropagation with Python

Let's go ahead and get started implementing backpropagation. Open up a new file, name it `neuralnetwork.py`, and let's get to work:

```
1  # import the necessary packages
2  import numpy as np
3
4  class NeuralNetwork:
5      def __init__(self, layers, alpha=0.1):
6          # initialize the list of weights matrices, then store the
7          # network architecture and learning rate
8          self.W = []
9          self.layers = layers
10         self.alpha = alpha
```

On **Line 2** we import the only required package we'll need for our implementation of backpropagation – the NumPy numerical processing library.

Line 5 then defines the constructor to our `NeuralNetwork` class. The constructor requires a single argument, followed a by a second optional one:

- `layers`: A list of integers which represents the actual *architecture* of the feedforward network. For example, a value of $[2,2,1]$ would imply that our first input layer has two nodes, our hidden layer has two nodes, and our final output layer has one node.
- `alpha`: Here we can specify the learning rate of our neural network. This value is applied during the weight update phase.

Line 8 initializes our list of weights for each layer, `W`. We then store `layers` and `alpha` on **Lines 9 and 10**.

Our weights list `W` is empty, so let's go ahead and initialize it now:

```
12          # start looping from the index of the first layer but
13          # stop before we reach the last two layers
14          for i in np.arange(0, len(layers) - 2):
15               # randomly initialize a weight matrix connecting the
16               # number of nodes in each respective layer together,
17               # adding an extra node for the bias
18               w = np.random.randn(layers[i] + 1, layers[i + 1] + 1)
19               self.W.append(w / np.sqrt(layers[i]))
```

On **Line 14** we start looping over the number of layers in the network (i.e., `len(layers)`), but we stop before the final two layer (we'll find out exactly why later in the explantation of this constructor).

Each layer in the network is randomly initialized by constructing an *MxN* weight matrix by sampling values from a standard, normal distribution (**Line 18**). The matrix is *MxN* since we wish to connect every node in *current layer* to every node in the *next layer*.

For example, let's suppose that `layers[i] = 2` and `layers[i + 1] = 2`. Our weight matrix would, therefore, be $2x2$ to connect all sets of nodes between the layers. However, we need to be careful here, as we are forgetting an important component – *the bias term*. To account for the bias, we add one to the number of `layers[i]` and `layers[i + 1]` – doing so changes our weight matrix `w` to have the shape $3x3$ given $2+1$ nodes for the current layer and $2+1$ nodes for the next layer. We scale `w` by dividing by the square root of the number of nodes in the current layer, thereby normalizing the variance of each neuron's output [57] (**Line 19**).

The final code block of the constructor handles the special where the input connections need a bias term, but the output does not:

```
21          # the last two layers are a special case where the input
22          # connections need a bias term but the output does not
23          w = np.random.randn(layers[-2] + 1, layers[-1])
24          self.W.append(w / np.sqrt(layers[-2]))
```

Again, these weight values are randomly sampled and then normalized.

The next function we define is a Python "magic method" named `__repr__` – this function is useful for debugging:

```
26     def __repr__(self):
27          # construct and return a string that represents the network
28          # architecture
29          return "NeuralNetwork: {}".format(
30               "-".join(str(l) for l in self.layers))
```

In our case, we'll format a string for our `NeuralNetwork` object by concatenating the integer value of the number of nodes in each layer. Given a `layers` value of `(2, 2, 1)`, the output of calling this function will be:

```
1  >>> from pyimagesearch.nn import NeuralNetwork
2  >>> nn = NeuralNetwork([2, 2, 1])
3  >>> print(nn)
4  NeuralNetwork: 2-2-1
```

Next, we can define our sigmoid activation function:

```
32      def sigmoid(self, x):
33          # compute and return the sigmoid activation value for a
34          # given input value
35          return 1.0 / (1 + np.exp(-x))
```

As well as the *derivative* of the sigmoid which we'll use during the backward pass:

```
37      def sigmoid_deriv(self, x):
38          # compute the derivative of the sigmoid function ASSUMING
39          # that 'x' has already been passed through the 'sigmoid'
40          # function
41          return x * (1 - x)
```

Again, note that whenever you perform backpropagation, you'll always want to choose an activation function that is *differentiable*.

We'll draw inspiration from the scikit-learn library and define a function named `fit` which will be responsible for actually training our `NeuralNetwork`:

```
43      def fit(self, X, y, epochs=1000, displayUpdate=100):
44          # insert a column of 1's as the last entry in the feature
45          # matrix -- this little trick allows us to treat the bias
46          # as a trainable parameter within the weight matrix
47          X = np.c_[X, np.ones((X.shape[0]))]
48
49          # loop over the desired number of epochs
50          for epoch in np.arange(0, epochs):
51              # loop over each individual data point and train
52              # our network on it
53              for (x, target) in zip(X, y):
54                  self.fit_partial(x, target)
55
56              # check to see if we should display a training update
57              if epoch == 0 or (epoch + 1) % displayUpdate == 0:
58                  loss = self.calculate_loss(X, y)
59                  print("[INFO] epoch={}, loss={:.7f}".format(
60                      epoch + 1, loss))
```

The `fit` method requires two parameters, followed by two optional ones. The first, `X`, is our *training data*. The second, `y`, is the corresponding class labels for each entry in `X`. We then specify `epochs`, which is the number of epochs we'll train our network for. The `displayUpdate` parameter simply controls how many *N* epochs we'll print training progress to our terminal.

On **Line 47** we perform the bias trick by inserting a column of 1's as the last entry in our feature matrix, `X`. From there, we start looping over our number of `epochs` on **Line 50**. For each epoch, we'll loop over each individual data point in our training set, make a prediction on the data point, compute the backpropagation phase, and then update our weight matrix (**Lines 53 and 54**). **Lines 57-60** simply check to see if we should display a training update to our terminal.

The actual heart of the backpropagation algorithm is found inside our `fit_partial` method below:

```
62        def fit_partial(self, x, y):
63            # construct our list of output activations for each layer
64            # as our data point flows through the network; the first
65            # activation is a special case -- it's just the input
66            # feature vector itself
67            A = [np.atleast_2d(x)]
```

The `fit_partial` function requires two parameters:

- `x`: An individual data point from our design matrix.
- `y`: The corresponding class label.

We then initialize a list, `A`, on **Line 67** – this list is responsible for storing the output activations for each layer as our data point `x` forward propagates through the network. We initialize this list with `x`, which is simply the input data point.

From here, we can start the forward propagation phase:

```
69            # FEEDFORWARD:
70            # loop over the layers in the network
71            for layer in np.arange(0, len(self.W)):
72                # feedforward the activation at the current layer by
73                # taking the dot product between the activation and
74                # the weight matrix -- this is called the "net input"
75                # to the current layer
76                net = A[layer].dot(self.W[layer])
77
78                # computing the "net output" is simply applying our
79                # nonlinear activation function to the net input
80                out = self.sigmoid(net)
81
82                # once we have the net output, add it to our list of
83                # activations
84                A.append(out)
```

We start looping over every layer in the network on **Line 71**. The *net input* to the current `layer` is computed by taking the dot product between the activation and the weight matrix (**Line 76**). The *net output* of the current layer is then computed by passing the net input through the nonlinear sigmoid activation function. Once we have the net output, we add it to our list of activations (**Line 84**).

Believe it or not, this code is the *entirety* of the forward pass described in Section 10.1.3 above – we are simply looping over each of the layers in the network, taking the dot product between the activation and the weights, passing the value through a nonlinear activation function, and continuing to the next layer. The final entry in `A` is thus the output of the last layer in our network (i.e., the *prediction*).

Now that the forward pass is done, we can move on to the slightly more complicated backward pass:

```
86            # BACKPROPAGATION
87            # the first phase of backpropagation is to compute the
88            # difference between our *prediction* (the final output
89            # activation in the activations list) and the true target
90            # value
```

```
91          error = A[-1] - y
92
93          # from here, we need to apply the chain rule and build our
94          # list of deltas 'D'; the first entry in the deltas is
95          # simply the error of the output layer times the derivative
96          # of our activation function for the output value
97          D = [error * self.sigmoid_deriv(A[-1])]
```

The first phase of the backward pass is to compute our `error`, or simply the difference between our *predicted* label and the *ground-truth* label (**Line 91**). Since the final entry in the activations list `A` contains the output of the network, we can access the output prediction via `A[-1]`. The value `y` is the target output for the input data point `x`.

When using the Python programming language, specifying an index value of `-1` indicates that we would like to access the *last* entry in the list. You can read more about Python array indexes and slices in this tutorial: http://pyimg.co/6dfae.

Next, we need to start applying the chain rule to build our list of deltas, `D`. The deltas will be used to update our weight matrices, scaled by the learning rate `alpha`. The first entry in the deltas list is the error of our output layer multiplied by the derivative of the sigmoid for the output value (**Line 97**).

Given the delta for the final layer in the network, we can now work backward using `for` loop:

```
99          # once you understand the chain rule it becomes super easy
100         # to implement with a 'for' loop -- simply loop over the
101         # layers in reverse order (ignoring the last two since we
102         # already have taken them into account)
103         for layer in np.arange(len(A) - 2, 0, -1):
104             # the delta for the current layer is equal to the delta
105             # of the *previous layer* dotted with the weight matrix
106             # of the current layer, followed by multiplying the delta
107             # by the derivative of the nonlinear activation function
108             # for the activations of the current layer
109             delta = D[-1].dot(self.W[layer].T)
110             delta = delta * self.sigmoid_deriv(A[layer])
111             D.append(delta)
```

On **Line 103** we start looping over each of the layers in the network (ignoring the previous two layers as they are already accounted for in **Line 97**) in *reverse order* as we need to work *backward* to compute the delta updates for each layer. The `delta` for the current layer is equal to the delta of the previous layer, `D[-1]` dotted with the weight matrix of the current layer (**Line 109**). To finish off the computation of the `delta`, we multiply it by passing the activation for the `layer` through our derivative of the sigmoid (**Line 110**). We then update the deltas `D` list with the `delta` we just computed (**Line 111**).

Looking at this block of code we can see that the backpropagation step is iterative – we are simply taking the delta from the *previous layer*, dotting it with the weights of the *current layer*, and then multiplying by the derivative of the activation. This process is repeated until we reach the first layer in the network.

Given our deltas list `D`, we can move on to the weight update phase:

```
113             # since we looped over our layers in reverse order we need to
114             # reverse the deltas
115             D = D[::-1]
116
117             # WEIGHT UPDATE PHASE
118             # loop over the layers
119             for layer in np.arange(0, len(self.W)):
120                 # update our weights by taking the dot product of the layer
121                 # activations with their respective deltas, then multiplying
122                 # this value by some small learning rate and adding to our
123                 # weight matrix -- this is where the actual "learning" takes
124                 # place
125                 self.W[layer] += -self.alpha * A[layer].T.dot(D[layer])
```

Keep in mind that during the backpropagation step we looped over our layers in *reverse* order. To perform our weight update phase, we'll simply *reverse* the ordering of entries in `D` so we can loop over each layer sequentially from 0 to N, the total number of layers in the network (**Line 115**).

Updating our actual weight matrix (i.e., where the actual "learning" takes place) is accomplished on **Line 125**, which is our gradient descent. We take the dot product of the current `layer` activation, `A[layer]` with the deltas of the current `layer`, `D[layer]` and multiple them by the learning rate, `alpha`. This value is added to the weight matrix for the current `layer`, `W[layer]`.

We repeat this process for all layers in the network. After performing the weight update phase, backpropagation is officially done.

Once our network is trained on a given dataset, we'll want to make predictions on the testing set, which can be accomplished via the `predict` method below:

```
127         def predict(self, X, addBias=True):
128             # initialize the output prediction as the input features -- this
129             # value will be (forward) propagated through the network to
130             # obtain the final prediction
131             p = np.atleast_2d(X)
132
133             # check to see if the bias column should be added
134             if addBias:
135                 # insert a column of 1's as the last entry in the feature
136                 # matrix (bias)
137                 p = np.c_[p, np.ones((p.shape[0]))]
138
139             # loop over our layers in the network
140             for layer in np.arange(0, len(self.W)):
141                 # computing the output prediction is as simple as taking
142                 # the dot product between the current activation value `p`
143                 # and the weight matrix associated with the current layer,
144                 # then passing this value through a nonlinear activation
145                 # function
146                 p = self.sigmoid(np.dot(p, self.W[layer]))
147
148             # return the predicted value
149             return p
```

The `predict` function is simply a glorified forward pass. This function accepts one required parameter followed by a second optional one:

- `X`: The data points we'll be predicting class labels for.

- `addBias`: A boolean indicating whether we need to add a column of 1's to `X` to perform the bias trick.

On **Line 131** we initialize `p`, the output predictions as the input data points `X`. This value `p` will be passed through every layer in the network, propagating until we reach the final output prediction.

On **Lines 134-137** we make a check to see if the bias term should be embedded into the data points. If so, we insert a column of 1's as the last column in the matrix (exactly as we did in the `fit` method above).

From there, we perform the forward propagation by looping over all layers in our network on **Line 140**. The data points `p` are updated by taking the dot product between the current activations `p` and the weight matrix for the current `layer`, followed by passing the output through our sigmoid activation function (**Line 146**).

Given that we are looping over all layers in the network, we'll eventually reach the final layer, which will give us our final class label prediction. We return the predicted value to the calling function on **Line 149**.

The final function we'll define inside the `NeuralNetwork` class will be used to calculate the loss across our *entire* training set:

```
151         def calculate_loss(self, X, targets):
152             # make predictions for the input data points then compute
153             # the loss
154             targets = np.atleast_2d(targets)
155             predictions = self.predict(X, addBias=False)
156             loss = 0.5 * np.sum((predictions - targets) ** 2)
157 
158             # return the loss
159             return loss
```

The `calculate_loss` function requires that we pass in the data points `X` along with their ground-truth labels, `targets`. We make predictions on `X` on **Line 155** and then compute the sum squared error on **Line 156**. The loss is then returned to the calling function on **Line 159**. As our network learns, we should see this loss decrease.

Backpropagation with Python Example #1: Bitwise XOR

Now that we have implemented our `NeuralNetwork` class, let's go ahead and train it on the bitwise XOR dataset. As we know from our work with the Perceptron, this dataset is *not* linearly separable – our goal will be to train a neural network that can model this nonlinear function.

Go ahead and open up a new file, name it `nn_xor.py`, and insert the following code:

```
1 # import the necessary packages
2 from pyimagesearch.nn import NeuralNetwork
3 import numpy as np
4 
5 # construct the XOR dataset
6 X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
7 y = np.array([[0], [1], [1], [0]])
```

Lines 2 and 3 import our required Python packages. Notice how we are importing our newly implemented `NeuralNetwork` class. **Lines 6 and 7** then construct the XOR dataset, as depicted by Table 10.1 earlier in this chapter.

We can now define our network architecture and train it:

```
9  # define our 2-2-1 neural network and train it
10 nn = NeuralNetwork([2, 2, 1], alpha=0.5)
11 nn.fit(X, y, epochs=20000)
```

On **Line 10** we instantiate our `NeuralNetwork` to have a $2-2-1$ architecture, implying there is:

1. An input layer with two nodes (i.e., our two inputs).
2. A single hidden layer with two nodes.
3. An output layer with one node.

Line 11 trains our network for a total of 20,000 epochs.

Once our network is trained, we'll loop over our XOR datasets, allow the network to predict the output for each one, and display the prediction to our screen:

```
13 # now that our network is trained, loop over the XOR data points
14 for (x, target) in zip(X, y):
15     # make a prediction on the data point and display the result
16     # to our console
17     pred = nn.predict(x)[0][0]
18     step = 1 if pred > 0.5 else 0
19     print("[INFO] data={}, ground-truth={}, pred={:.4f}, step={}".format(
20         x, target[0], pred, step))
```

Line 18 applies a step function to the sigmoid output. If the prediction is > 0.5, we'll return *one*, otherwise, we will return *zero*. Applying this step function allows us to binarize our output class labels, just like the XOR function.

To train our neural network using backpropagation with Python, simply execute the following command:

```
$ python nn_xor.py
[INFO] epoch=1, loss=0.5092796
[INFO] epoch=100, loss=0.4923591
[INFO] epoch=200, loss=0.4677865
...
[INFO] epoch=19800, loss=0.0002478
[INFO] epoch=19900, loss=0.0002465
[INFO] epoch=20000, loss=0.0002452
```

A plot of the squared loss is displayed below (Figure 10.11). As we can see, loss slowly decreases to approximately zero over the course of training. Furthermore, looking at the final four lines of the output we can see our predictions:

```
[INFO] data=[0 0], ground-truth=0, pred=0.0054, step=0
[INFO] data=[0 1], ground-truth=1, pred=0.9894, step=1
[INFO] data=[1 0], ground-truth=1, pred=0.9876, step=1
[INFO] data=[1 1], ground-truth=0, pred=0.0140, step=0
```

For each and every data point, our neural network was able to correctly learn the XOR pattern, demonstrating that our multi-layer neural network is capable of learning nonlinear functions.

To demonstrate that least one hidden layer is required to learn the XOR function, go back to **Line 10** where we define the $2-2-1$ architecture:

Figure 10.11: Loss over time for our $2-2-1$ neural network.

```
# define our 2-2-1 neural network and train it
nn = NeuralNetwork([2, 2, 1], alpha=0.5)
nn.fit(X, y, epochs=20000)
```

And change it to be a 2-1 architecture:

```
define our 2-1 neural network and train it
nn = NeuralNetwork([2, 1], alpha=0.5)
nn.fit(X, y, epochs=20000)
```

From there, you can attempt to retrain your network:

```
$ python nn_xor.py
...
[INFO] data=[0 0], ground-truth=0, pred=0.5161, step=1
[INFO] data=[0 1], ground-truth=1, pred=0.5000, step=1
[INFO] data=[1 0], ground-truth=1, pred=0.4839, step=0
[INFO] data=[1 1], ground-truth=0, pred=0.4678, step=0
```

No matter how much you fiddle with the learning rate or weight initializations, you'll never be able to approximate the XOR function. This fact is why multi-layer networks with nonlinear activation functions trained via backpropagation are so important – they enable us to learn patterns in datasets that are otherwise nonlinearly separable.

Backpropagation with Python Example: MNIST Sample

As a second, more interesting example, let's examine a subset of the MNIST dataset (Figure 10.12) for handwritten digit recognition. This subset of the MNIST dataset is built-into the scikit-learn library and includes 1,797 example digits, each of which are 8×8 grayscale images (the original images are 28×28. When flattened, these images are represented by an $8 \times 8 = 64$-dim vector.

Figure 10.12: A sample of the MNIST dataset. The goal of this dataset is to correctly classify the handwritten digits, $0 - 9$.

Let's go ahead and train our `NeuralNetwork` implementation on this MNIST subset now. Open up a new file, name it `nn_mnist.py`, and we'll get to work:

```
1  # import the necessary packages
2  from pyimagesearch.nn import NeuralNetwork
3  from sklearn.preprocessing import LabelBinarizer
4  from sklearn.model_selection import train_test_split
5  from sklearn.metrics import classification_report
6  from sklearn import datasets
```

We start on **Lines 2-6** by importing our required Python packages.

From there, we load the MNIST dataset from disk using the scikit-learn helper functions:

```
8  # load the MNIST dataset and apply min/max scaling to scale the
9  # pixel intensity values to the range [0, 1] (each image is
10 # represented by an 8 x 8 = 64-dim feature vector)
11 print("[INFO] loading MNIST (sample) dataset...")
12 digits = datasets.load_digits()
13 data = digits.data.astype("float")
14 data = (data - data.min()) / (data.max() - data.min())
15 print("[INFO] samples: {}, dim: {}".format(data.shape[0],
16       data.shape[1]))
```

We also perform min/max normalizing by scaling each digit into the range $[0, 1]$ (**Line 14**).

Next, let's construct a training and testing split, using 75% of the data for testing and 25% for evaluation:

```
18 # construct the training and testing splits
19 (trainX, testX, trainY, testY) = train_test_split(data,
20       digits.target, test_size=0.25)
21
22 # convert the labels from integers to vectors
23 trainY = LabelBinarizer().fit_transform(trainY)
24 testY = LabelBinarizer().fit_transform(testY)
```

We'll also encode our class label integers as vectors, a process called *one-hot encoding* that we will discuss in detail later in this chapter.

From there, we are ready to train our network:

```
26 # train the network
27 print("[INFO] training network...")
28 nn = NeuralNetwork([trainX.shape[1], 32, 16, 10])
29 print("[INFO] {}".format(nn))
30 nn.fit(trainX, trainY, epochs=1000)
```

Here we can see that we are training a `NeuralNetwork` with a $64 - 32 - 16 - 10$ architecture. The output layer has ten nodes due to the fact that there are ten possible output classes for the digits *0-9*.

We then allow our network to train for 1,000 epochs. Once our network has been trained, we can evaluate it on the testing set:

```
32 # evaluate the network
33 print("[INFO] evaluating network...")
34 predictions = nn.predict(testX)
35 predictions = predictions.argmax(axis=1)
36 print(classification_report(testY.argmax(axis=1), predictions))
```

Line 34 computes the output `predictions` for every data point in `testX`. The `predictions` array has the shape `(450, 10)` as there are 450 data points in the testing set, each of which with ten possible class label probabilities.

To find the class label with the *largest probability* for *each data point*, we use the `argmax` function on **Line 35** – this function will return the index of the label with the highest predicted probability. We then display a nicely formatted classification report to our screen on **Line 36**.

To train our custom `NeuralNetwork` implementation on the MNIST dataset, just execute the following command:

```
$ python nn_mnist.py
[INFO] loading MNIST (sample) dataset...
[INFO] samples: 1797, dim: 64
[INFO] training network...
[INFO] NeuralNetwork: 64-32-16-10
[INFO] epoch=1, loss=604.5868589
[INFO] epoch=100, loss=9.1163376
[INFO] epoch=200, loss=3.7157723
[INFO] epoch=300, loss=2.6078803
[INFO] epoch=400, loss=2.3823153
[INFO] epoch=500, loss=1.8420944
[INFO] epoch=600, loss=1.3214138
[INFO] epoch=700, loss=1.2095033
[INFO] epoch=800, loss=1.1663942
[INFO] epoch=900, loss=1.1394731
[INFO] epoch=1000, loss=1.1203779
[INFO] evaluating network...
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        45
```

```
             1       0.98      1.00      0.99        51
             2       0.98      1.00      0.99        47
             3       0.98      0.93      0.95        43
             4       0.95      1.00      0.97        39
             5       0.94      0.97      0.96        35
             6       1.00      1.00      1.00        53
             7       1.00      1.00      1.00        49
             8       0.97      0.95      0.96        41
             9       1.00      0.96      0.98        47

avg / total       0.98      0.98      0.98       450
```

Figure 10.13: Plotting training loss on the MNIST dataset using a $64 - 32 - 16 - 10$ feedforward neural network.

I have included a plot of the squared loss as well (Figure 10.13). Notice how our loss starts off very high, but quickly drops during the training process. Our classification report demonstrates that we are obtaining $\approx 98\%$ classification accuracy on our testing set; however, we are having some trouble classifying digits 4 and 5 (95% and 94% accuracy, respectively). Later in this book, we'll learn how to train Convolutional Neural Networks on the *full* MNIST dataset and improve our accuracy further.

Backpropagation Summary

In this section, we learned how to implement the backpropagation algorithm from scratch using Python. Backpropagation is a generalization of the gradient descent family of algorithms that is specifically used to train multi-layer feedforward networks.

The backpropagation algorithm consists of two phases:

1. The forward pass where we pass our inputs through the network to obtain our output classifications.

2. The backward pass (i.e., weight update phase) where we compute the gradient of the loss function and use this information to iteratively apply the chain rule to update the weights in our network.

Regardless of whether we are working with simple feedforward neural networks or complex, deep Convolutional Neural Networks, the backpropagation algorithm is still used to train these models. This is accomplished by ensuring that the activation functions inside the network are *differentiable*, allowing the chain rule to be applied. Furthermore, any other layers inside the network that require updates to their weights/parameters, must also be compatible with backpropagation as well.

We implemented our backpropagation algorithm using the Python programming language and devised a multi-layer, feedforward `NeuralNetwork` class. This implementation was then trained on the XOR dataset to demonstrate that our neural network is capable of learning nonlinear functions by applying the backpropagation algorithm with at least one hidden layer. We then applied the same backpropagation + Python implementation to a subset of the MNIST dataset to demonstrate that the algorithm can be used to work with image data as well.

In practice, backpropagation can be not only challenging to implement (due to bugs in computing the gradient), but also hard to make efficient without special optimization libraries, which is why we often use libraries such as Keras, TensorFlow, and mxnet that have *already* (correctly) implemented backpropagation using optimized strategies.

10.1.4 Multi-layer Networks with Keras

Now that we have implemented neural networks in *pure Python*, let's move on to the preferred implementation method – using a dedicated (highly optimized) neural network library such as Keras.

In the next two sections, I'll discuss how to implement feedforward, multi-layer networks and apply them to the MNIST and CIFAR-10 datasets. These result will hardly be "state-of-the-art", but will serve two purposes:

- To demonstrate how you can implement simple neural networks using the Keras library.
- Obtain a baseline using standard neural networks which we will later compare to Convolutional Neural Networks (noting that CNNS will *dramatically* outperform our previous methods).

MNIST

Section 10.1.3 above, we only used a sample of the MNIST dataset for two reasons:

- To demonstrate how to implement your first feedforward neural network in pure Python.
- To facilitate faster result gathering – given that our pure Python implementation is by definition unoptimized, it will take longer to run.

Therefore, we used a *sample* of the dataset. In this section, we'll be using the *full* MNIST dataset, consisting of 70,000 data points (7,000 examples per digit). Each data point is represented by a 784-d vector, corresponding to the (flattened) 28×28 images in the MNIST dataset. Our goal is to train a neural network (using Keras) to obtain $> 90\%$ accuracy on this dataset.

As we'll find out, using Keras to build our network architecture is *substantially easier* than our pure Python version. In fact, the actual network architecture will only occupy *four lines of code* – the rest of the code in this example simply involves loading the data from disk, transforming the class labels, and then displaying the results.

To get started, open up a new file, name it `keras_mnist.py`, and insert the following code:

```
1 # import the necessary packages
2 from sklearn.preprocessing import LabelBinarizer
```

```
3  from sklearn.model_selection import train_test_split
4  from sklearn.metrics import classification_report
5  from keras.models import Sequential
6  from keras.layers.core import Dense
7  from keras.optimizers import SGD
8  from sklearn import datasets
9  import matplotlib.pyplot as plt
10 import numpy as np
11 import argparse
```

Lines 2-11 import our required Python packages. The `LabelBinarizer` will be used to one-hot encode our *integer labels* as *vector labels*. One-hot encoding transforms categorical labels from a single integer to a vector. Many machine learning algorithms (including neural networks) benefit from this type of label representation. I'll be discussing one-hot encoding in more detail and providing multiple examples (including using the `LabelBinarizer`) later in this section.

The `train_test_split` on **Line 3** will be used to create our training and testing splits from the MNIST dataset. The `classification_report` function will give us a nicely formatted report displaying the total accuracy of our model, along with a breakdown on the classification accuracy for *each* digit.

Lines 5-7 import the necessary packages to create a simple feedforward neural network with Keras. The `Sequential` class indicates that our network will be feedforward and layers will be added to the class *sequentially*, one on top of the other. The `Dense` class on **Line 6** is the implementation of our fully-connected layers. For our network to actually learn, we need to apply `SGD` (**Line 7**) to optimize the parameters of the network. Finally, to gain access to full MNIST dataset, we need to import the `datasets` helper from scikit-learn on **Line 8**.

Let's move on to parsing our command line arguments:

```
13 # construct the argument parse and parse the arguments
14 ap = argparse.ArgumentParser()
15 ap.add_argument("-o", "--output", required=True,
16     help="path to the output loss/accuracy plot")
17 args = vars(ap.parse_args())
```

We only need a single switch here, `--output`, which is the path to where our figure plotting the loss and accuracy over time will be saved to disk.

Next, let's load the full MNIST dataset:

```
19 # grab the MNIST dataset (if this is your first time running this
20 # script, the download may take a minute -- the 55MB MNIST dataset
21 # will be downloaded)
22 print("[INFO] loading MNIST (full) dataset...")
23 dataset = datasets.fetch_mldata("MNIST Original")
24
25 # scale the raw pixel intensities to the range [0, 1.0], then
26 # construct the training and testing splits
27 data = dataset.data.astype("float") / 255.0
28 (trainX, testX, trainY, testY) = train_test_split(data,
29     dataset.target, test_size=0.25)
```

Line 23 loads the MNIST dataset from disk. If you have *never* run this function before, then the MNIST dataset will be downloaded and stored locally to your machine – this download is 55MB

and may take a minute or two to finish downloading, depending on your internet connection. Once the dataset has been downloaded, it is cached to your machine and will not have to be downloaded again.

We then perform data normalization on **Line 27** by scaling the pixel intensities to the range $[0, 1]$. We create a training and testing split, using 75% of the data for training and 25% for testing on **Lines 28 and 29**.

Given the training and testing splits, we can now encode our labels:

```
31 # convert the labels from integers to vectors
32 lb = LabelBinarizer()
33 trainY = lb.fit_transform(trainY)
34 testY = lb.transform(testY)
```

Each data point in the MNIST dataset has an integer label in the range $[0, 9]$, one for each of the possible ten digits in the MNIST dataset. A label with a value of `0` indicates that the corresponding image contains a zero digit. Similarly, a label with a value of `8` indicates that the corresponding image contains the number eight.

However, we first need to transform these *integer labels* into *vector labels*, where the index in the vector for label is set to `1` and `0` otherwise (this process is called *one-hot encoding*).

For example, consider the label `3` and we wish to binarize/one-hot encode it – the label `3` now becomes:

```
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
```

Notice how only the index for the digit three is set to one – all other entries in the vector are set to zero. Astute readers may wonder why the *fourth* and not the *third* entry in the vector is updated? Recall that the first entry in the label is actually for the digit zero. Therefore, the entry for the digit three is actually the fourth index in the list.

Here is a second example, this time with the label `1` binarized:

```
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
```

The second entry in the vector is set to one (since the first entry corresponds to the label `0`), while all other entries are set to zero.

I have included the one-hot encoding representations for each digit, $0 - 9$, in the listing below:

```
0: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
1: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
2: [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
3: [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
4: [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
5: [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
6: [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
7: [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
8: [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
9: [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
```

This encoding may seem tedious, but many machine learning algorithms (including neural networks), benefit from this label representation. Luckily, most machine learning software packages provide a method/function to perform one-hot encoding, removing much of the tediousness.

Lines 32-34 simply perform this process of one-hot encoding the input *integer labels* as *vector labels* for both the training and testing set.

Next, let's define our network architecture:

```
36 # define the 784-256-128-10 architecture using Keras
37 model = Sequential()
38 model.add(Dense(256, input_shape=(784,), activation="sigmoid"))
39 model.add(Dense(128, activation="sigmoid"))
40 model.add(Dense(10, activation="softmax"))
```

As you can see, our network is a feedforward architecture, instantiated by the `Sequential` class on **Line 37** – this architecture implies that the layers will be stacked on top of each other with the output of the previous layer feeding into the next.

Line 38 defines the first fully-connected layer in the network. The `input_shape` is set to `784`, the dimensionality of each MNIST data points. We then learn 256 weights in this layer and apply the sigmoid activation function. The next layer (**Line 39**) learns 128 weights. Finally, **Line 40** applies another fully-connected layer, this time only learning 10 weights, corresponding to the ten (0-9) output classes. Instead of a sigmoid activation, we'll use a softmax activation to obtain normalized class probabilities for each prediction.

Let's go ahead and train our network:

```
42 # train the model using SGD
43 print("[INFO] training network...")
44 sgd = SGD(0.01)
45 model.compile(loss="categorical_crossentropy", optimizer=sgd,
46     metrics=["accuracy"])
47 H = model.fit(trainX, trainY, validation_data=(testX, testY),
48     epochs=100, batch_size=128)
```

On **Line 44** we initialize the SGD optimizer with a learning rate of `0.01` (which we may commonly write as `1e-2`). We'll use the category cross-entropy loss function as our loss metric (**Lines 45 and 46**). Using the cross-entropy loss function is also why we had to convert our integer labels to vector labels.

A call to `.fit` of the `model` on **Line 47 and 48** kicks off the training of our neural network. We'll supply the training data and training labels as the first two arguments to the method.

The `validation_data` can then be supplied, which is our testing split. In *most* circumstances, such as when you are tuning hyperparameters or deciding on a model architecture, you'll want your validation set to be a *true* validation set and not your testing data. In this case, we are simply demonstrating how to train a neural network from scratch using Keras so we're being a bit lenient with our guidelines. Future chapters in this book, as well as the more advanced content in the *Practitioner Bundle* and *ImageNet Bundle*, are *much* more rigorous in the scientific method; however, for now, simply focus on the code and grasp how the network is trained.

We'll allow our network to train for a total of 100 epochs using a batch size of 128 data points at a time. The method returns a dictionary, `H`, which we'll use to plot the loss/accuracy of the network overtime in a couple of code blocks.

Once the network has finished training, we'll want to evaluate it on the testing data to obtain our final classifications:

```
50 # evaluate the network
51 print("[INFO] evaluating network...")
52 predictions = model.predict(testX, batch_size=128)
53 print(classification_report(testY.argmax(axis=1),
54      predictions.argmax(axis=1),
55      target_names=[str(x) for x in lb.classes_]))
```

A call to the `.predict` method of `model` will return the class label probabilities for *every* data point in `testX` (**Line 52**). Thus, if you were to inspect the `predictions` NumPy array it would have the shape `(X, 10)` as there are 17,500 total data points in the testing set and ten possible class labels (the digits 0-9).

Each entry in a given row is, therefore, a *probability*. To determine the class with the *largest* probability, we can simply call `.argmax(axis=1)` as we do on **Line 53**, which will give us the *index* of the class label with the largest probability, and, therefore, our final output classification. The final output classification by the network is tabulated, and then a final classification report is displayed to our console on **Lines 53-55**.

Our final code block handles plotting the training loss, training accuracy, validation loss, and validation accuracy over time:

```
57 # plot the training loss and accuracy
58 plt.style.use("ggplot")
59 plt.figure()
60 plt.plot(np.arange(0, 100), H.history["loss"], label="train_loss")
61 plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss")
62 plt.plot(np.arange(0, 100), H.history["acc"], label="train_acc")
63 plt.plot(np.arange(0, 100), H.history["val_acc"], label="val_acc")
64 plt.title("Training Loss and Accuracy")
65 plt.xlabel("Epoch #")
66 plt.ylabel("Loss/Accuracy")
67 plt.legend()
68 plt.savefig(args["output"])
```

This plot is then saved to disk based on the `--output` command line argument.

To train our network of fully-connected layers on MNIST, just execute the following command:

```
$ python keras_mnist.py --output output/keras_mnist.png
[INFO] loading MNIST (full) dataset...
[INFO] training network...
Train on 52500 samples, validate on 17500 samples
Epoch 1/100
1s - loss: 2.2997 - acc: 0.1088 - val_loss: 2.2918 - val_acc: 0.1145
Epoch 2/100
1s - loss: 2.2866 - acc: 0.1133 - val_loss: 2.2796 - val_acc: 0.1233
Epoch 3/100
1s - loss: 2.2721 - acc: 0.1437 - val_loss: 2.2620 - val_acc: 0.1962
...
Epoch 98/100
1s - loss: 0.2811 - acc: 0.9199 - val_loss: 0.2857 - val_acc: 0.9153
Epoch 99/100
1s - loss: 0.2802 - acc: 0.9201 - val_loss: 0.2862 - val_acc: 0.9148
Epoch 100/100
```

```
1s - loss: 0.2792 - acc: 0.9204 - val_loss: 0.2844 - val_acc: 0.9160
[INFO] evaluating network...
              precision    recall  f1-score   support

         0.0       0.94      0.96      0.95      1726
         1.0       0.95      0.97      0.96      2004
         2.0       0.91      0.89      0.90      1747
         3.0       0.91      0.88      0.89      1828
         4.0       0.91      0.93      0.92      1686
         5.0       0.89      0.86      0.88      1581
         6.0       0.92      0.96      0.94      1700
         7.0       0.92      0.94      0.93      1814
         8.0       0.88      0.88      0.88      1679
         9.0       0.90      0.88      0.89      1735

avg / total       0.92      0.92      0.92     17500
```

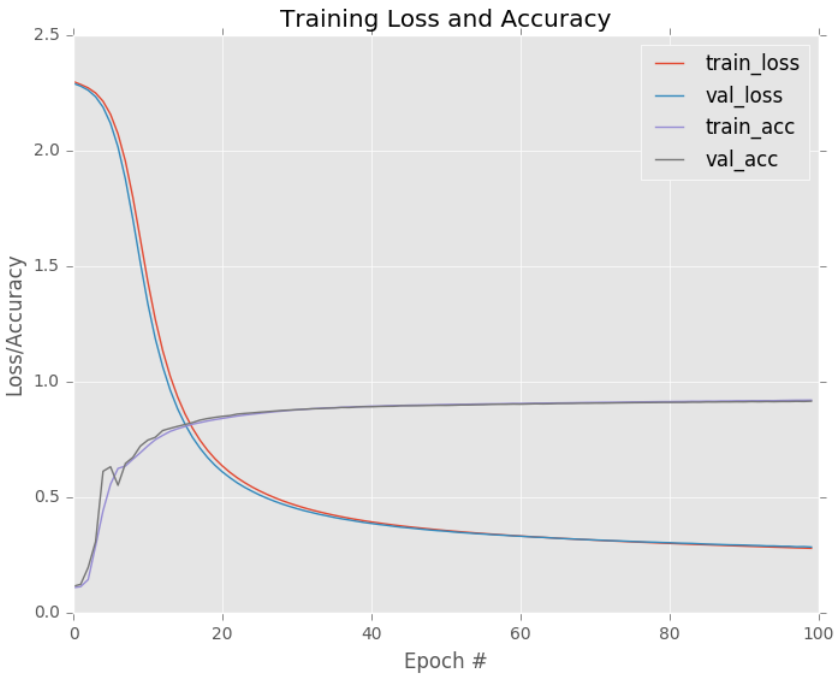


Figure 10.14: Training a $784 - 256 - 128 - 10$ feedforward neural network with Keras on the *full* MNIST dataset. Notice how our training and validation curves are near identical, implying there is no overfitting occurring.

As the results demonstrate, we are obtaining $\approx 92\%$ accuracy. Furthermore, the training and validation curves match each other *nearly identically* (Figure 10.14), indicating there is no overfitting or issues with the training process.

In fact, if you are unfamiliar with the MNIST dataset, you might think 92% accuracy is *excellent* – and it was, perhaps 20 years ago. As we'll find out in Chapter 14, using Convolutional Neural Networks, we can easily obtain $> 98\%$ accuracy. Current state-of-the-art approaches can even break 99% accuracy.

While on the surface it may appear that our (strictly) fully-connected network is performing well, we can actually do much better. And as we'll see in the next section, strictly fully-connected networks applied to more challenging datasets can in some cases do just barely better than guessing randomly.

CIFAR-10

When it comes to computer vision and machine learning, the MNIST dataset is the classic definition of a "benchmark" dataset, one that is too easy to obtain high accuracy results on, and not representative of the images we'll see in the real world.

For a more challenging benchmark dataset, we commonly use CIFAR-10, a collection of 60,000, 32×32 RGB images, thus implying that each image in the dataset is represented by $32 \times 32 \times 3 = 3,072$ integers. As the name suggests, CIFAR-10 consists of 10 classes, including *airplane*, *automobile*, *bird*, *cat*, *deer*, *dog*, *frog*, *horse*, *ship*, and *truck*. A sample of the CIFAR-10 dataset for each class can be seen in Figure 10.15.

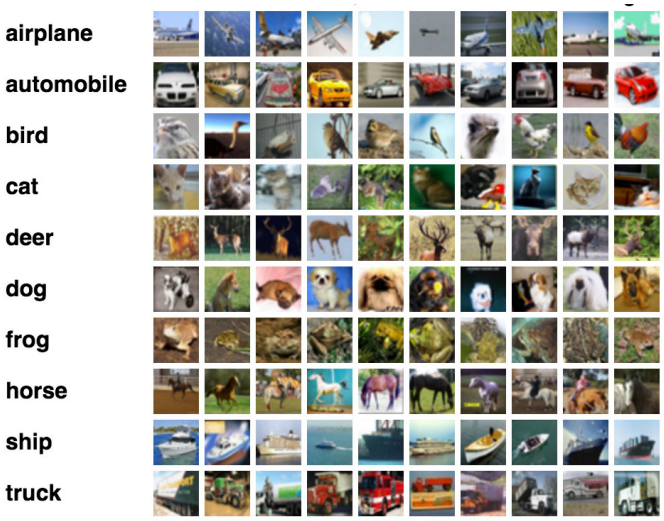


Figure 10.15: Example images from the ten class CIFAR-10 dataset.

Each class is evenly represented with 6,000 images per class. When training and evaluating a machine learning model on CIFAR-10, it's typical to use the predefined data splits by the authors and use 50,000 images for training and 10,000 for testing.

CIFAR-10 is *substantially harder* than the MNIST dataset. The challenge comes from the dramatic variance in how objects appear. For example, we can no longer assume that an image containing a green pixel at a given (x, y)-coordinate is a frog. This pixel could be a background of a forest that contains a deer. Or it could be the color of a green car or truck.

These assumptions are a stark contrast to the MNIST dataset, where the network can learn assumptions regarding the spatial distribution of pixel intensities. For example, the spatial distribution of foreground pixels of a 1 is substantially different than a 0 or a 5. This type of variance in object appearance makes applying a series of fully-connected layers much more challenging. As we'll find out in the rest of this section, standard `FC` (fully-connected) layer networks are not suited for this type of image classification.

Let's go ahead and get started. Open up a new file, name it `keras_cifar10.py`, and insert the following code:

```
1  # import the necessary packages
2  from sklearn.preprocessing import LabelBinarizer
3  from sklearn.metrics import classification_report
4  from keras.models import Sequential
5  from keras.layers.core import Dense
6  from keras.optimizers import SGD
7  from keras.datasets import cifar10
8  import matplotlib.pyplot as plt
9  import numpy as np
10 import argparse
```

Lines 2-10 import our required Python packages to build our fully-connected network, identical to the previous section with MNIST. The exception is the special utility function on **Line 7** – since CIFAR-10 is such a common dataset that researchers benchmark machine learning and deep learning algorithms on, it's common to see deep learning libraries provide simple helper functions to *automatically* load this dataset from disk.

Next, we can parse our command line arguments:

```
16 # construct the argument parse and parse the arguments
17 ap = argparse.ArgumentParser()
18 ap.add_argument("-o", "--output", required=True,
19     help="path to the output loss/accuracy plot")
20 args = vars(ap.parse_args())
```

The only command line argument we need is `--output`, the path to our output loss/accuracy plot.

Let's go ahead and load the CIFAR-10 dataset:

```
18 # load the training and testing data, scale it into the range [0, 1],
19 # then reshape the design matrix
20 print("[INFO] loading CIFAR-10 data...")
21 ((trainX, trainY), (testX, testY)) = cifar10.load_data()
22 trainX = trainX.astype("float") / 255.0
23 testX = testX.astype("float") / 255.0
24 trainX = trainX.reshape((trainX.shape[0], 3072))
25 testX = testX.reshape((testX.shape[0], 3072))
```

A call to `cifar10.load_data` on **Line 21** automatically loads the CIFAR-10 dataset from disk, pre-segmented into training and testing split. If this is the *first* time you are calling `cifar10.load_data`, then this function will fetch and download the dataset for you. This file is $\approx 170MB$, so be patient as it is downloaded and unarchived. Once the file is downloaded once, it will be cached locally on your machine and will not have to be downloaded again.

Lines 22 and 23 convert the data type CIFAR-10 from unsigned 8-bit integers to floating point, followed by scaling the data to the range $[0, 1]$. **Lines 24 and 25** are responsible for *reshaping* the design matrix for the training and testing data. Recall that each image in the CIFAR-10 dataset is represented by a $32 \times 32 \times 3$ image.

For example, `trainX` has the shape `(50000, 32, 32, 3)` and `testX` has the shape `(10000, 32, 32, 3)`. If we were to *flatten* this image into a single list of floating point values, the list would have a total of $32 \times 32 \times 3 = 3,072$ total entries in it.

To flatten each of the images in the training and testing sets, we simply use the `.reshape` function of NumPy. After this function executes, `trainX` now has the shape `(50000, 3072)` while `testX` has the shape `(10000, 3072)`.

Now that the CIFAR-10 dataset has been loaded from disk, let's once again binarize the class label integers into vectors, followed by initializing a list of the actual *names* of the class labels:

```
27 # convert the labels from integers to vectors
28 lb = LabelBinarizer()
29 trainY = lb.fit_transform(trainY)
30 testY = lb.transform(testY)
31
32 # initialize the label names for the CIFAR-10 dataset
33 labelNames = ["airplane", "automobile", "bird", "cat", "deer",
34     "dog", "frog", "horse", "ship", "truck"]
```

It's now time to define the network architecture:

```
36 # define the 3072-1024-512-10 architecture using Keras
37 model = Sequential()
38 model.add(Dense(1024, input_shape=(3072,), activation="relu"))
39 model.add(Dense(512, activation="relu"))
40 model.add(Dense(10, activation="softmax"))
```

Line 37 instantiates the `Sequential` class. We then add the first `Dense` layer which has an `input_shape` of 3072, a node for each of the 3,072 flattened pixel values in the design matrix – this layer is then responsible for learning 1,024 weights. We'll also swap out the antiquated sigmoid for a ReLU activation in hopes of improve network performance.

The next fully-connected layer (**Line 39**) learns 512 weights, while the final layer (**Line 40**) learns weights corresponding to ten possible output classifications, along with a softmax classifier to obtain the final output probabilities for each class.

Now that the architecture of the network is defined, we can train it:

```
42 # train the model using SGD
43 print("[INFO] training network...")
44 sgd = SGD(0.01)
45 model.compile(loss="categorical_crossentropy", optimizer=sgd,
46     metrics=["accuracy"])
47 H = model.fit(trainX, trainY, validation_data=(testX, testY),
48     epochs=100, batch_size=32)
```

We'll use the SGD optimizer to train the network with a learning rate of `0.01`, a fairly standard initial choice. The network will be trained for a total of 100 epochs using batches of 32.

Once the network has been trained, we can evaluate it using `classification_report` to obtain a more detailed review of model performance:

```
50 # evaluate the network
51 print("[INFO] evaluating network...")
52 predictions = model.predict(testX, batch_size=32)
53 print(classification_report(testY.argmax(axis=1),
54     predictions.argmax(axis=1), target_names=labelNames))
```

And finally, we'll also plot the loss/accuracy over time as well:

```
# plot the training loss and accuracy
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, 100), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, 100), H.history["acc"], label="train_acc")
plt.plot(np.arange(0, 100), H.history["val_acc"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.savefig(args["output"])
```

To train our network on CIFAR-10, open up a terminal and execute the following command:

```
$ python keras_cifar10.py --output output/keras_cifar10.png
[INFO] training network...
Train on 50000 samples, validate on 10000 samples
Epoch 1/100
7s - loss: 1.8409 - acc: 0.3428 - val_loss: 1.6965 - val_acc: 0.4070
Epoch 2/100
7s - loss: 1.6537 - acc: 0.4160 - val_loss: 1.6561 - val_acc: 0.4163
Epoch 3/100
7s - loss: 1.5701 - acc: 0.4449 - val_loss: 1.6049 - val_acc: 0.4376
...
Epoch 98/100
7s - loss: 0.0292 - acc: 0.9969 - val_loss: 2.2477 - val_acc: 0.5712
Epoch 99/100
7s - loss: 0.0272 - acc: 0.9972 - val_loss: 2.2514 - val_acc: 0.5717
Epoch 100/100
7s - loss: 0.0252 - acc: 0.9976 - val_loss: 2.2492 - val_acc: 0.5739
[INFO] evaluating network...
             precision    recall  f1-score   support

   airplane       0.63      0.66      0.64      1000
 automobile       0.69      0.65      0.67      1000
       bird       0.48      0.43      0.45      1000
        cat       0.40      0.38      0.39      1000
       deer       0.52      0.51      0.51      1000
        dog       0.48      0.47      0.48      1000
       frog       0.64      0.63      0.64      1000
      horse       0.63      0.62      0.63      1000
       ship       0.64      0.74      0.69      1000
      truck       0.59      0.65      0.62      1000

avg / total       0.57      0.57      0.57     10000
```

Looking at the output, you can see that our network obtained 57% accuracy. Examining our plot of loss and accuracy over time (Figure 10.16), we can see that our network struggles with overfitting past epoch 10. Loss initially starts to decrease, levels out a bit, and then skyrockets, and never comes down again. All the while training loss is falling consistently epoch-over-epoch.

Figure 10.16: Using a standard feedforward neural network leads to dramatic overfitting in the more challenging CIFAR-10 dataset (notice how training loss falls while validation loss rises dramatically). To be successful at the CIFAR-10 challenge, we'll need a powerful technique – Convolutional Neural Networks.

This behavior of *decreasing* training loss while validation loss *increases* is indicative of *extreme overfitting*.

We could certainly consider optimizing our hyperparameters further, in particular, experimenting with varying learning rates and increasing both the depth and the number of nodes in the network, but we would be fighting for meager gains.

The fact is this – basic feedforward networks with strictly fully-connected layers are not suitable for challenging image datasets. For that, we need a more advanced approach: Convolutional Neural Networks. Luckily, CNNs are the topic of the entire remainder of this book. By the time you finish the *Starter Bundle*, you'll be able to obtain over 79% accuracy on CIFAR-10. If you so choose to study deep learning in more depth, the *Practitioner Bundle* will demonstrate how to increase your accuracy to over 93%, putting us in the league of state-of-the-art results [110].

10.1.5 The Four Ingredients in a Neural Network Recipe

You might have started to notice a pattern in our Python code examples when training neural networks. There are four main ingredients you need to put together your own neural network and deep learning algorithm: a *dataset*, a *model/architecture*, a *loss function*, and an *optimization method*. We'll review each of these ingredients below.

Dataset

The dataset is the first ingredient in training a neural network – the data itself along with the problem we are trying to solve define our end goals. For example, are we using neural networks to perform a regression analysis to predict the value of homes in a specific suburb in 20 years? Is our goal

to perform unsupervised learning, such as dimensionality reduction? Or are we trying to perform classification?

In the context of this book, we're strictly on *image classification*; however, the combination of your dataset and the problem you are trying to solve influences your choice in loss function, network architecture, and optimization method used to train the model. Usually, we have little choice in our dataset (unless you're working on a hobby project) – we are given a dataset with some expectation on what the results from our project should be. It is then up to us to train a machine learning model on the dataset to perform well on the given task.

Loss Function

Given our dataset and target goal, we need to define a loss function that aligns with the problem we are trying to solve. In nearly all image classification problems using deep learning, we'll be using cross-entropy loss. For > 2 classes we call this *categorical cross-entropy*. For two class problems, we call the loss *binary cross-entropy*.

Model/Architecture

Your network architecture can be considered the first actual "choice" you have to make as an ingredient. Your dataset is likely chosen for you (or at least you've *decided* that you want to work with a given dataset). And if you're performing classification, you'll in all likelihood be using cross-entropy as your loss function.

However, your network architecture can vary dramatically, especially when with which optimization method you choose to train your network. After taking the time to explore your dataset and look at:

1. How many data points you have.
2. The number of classes.
3. How similar/dissimilar the classes are.
4. The intra-class variance.

You should start to develop a "feel" for a network architecture you are going to use. This takes practice as deep learning is part science, part art – in fact, the rest of this book is dedicated to helping you develop both of these skills.

Keep in mind that the number of layers and nodes in your network architecture (along with any type of regularization) is likely to change as you perform more and more experiments. The more results you gather, the better equipped you are to make informed decisions on which techniques to try next.

Optimization Method

The final ingredient is to define an optimization method. As we've seen thus far in this book *Stochastic Gradient Descent* (Section 9.2) is used quite often. Other optimization methods exist, including RMSprop [90], Adagrad [111], Adadelta [112], and Adam [113]; however, these are more advanced optimization methods that we'll cover in the *Practitioner Bundle*.

Even despite all these newer optimization methods, SGD is *still* the workhorse of deep learning – most neural networks are trained via SGD, including the networks obtaining state-of-the-art accuracy on challenging image datasets such as ImageNet.

When training deep learning networks, especially when you're first getting started and learning the ropes, *SGD should be your optimizer of choice*. You then need to set a proper learning rate and regularization strength, the total number of epochs the network should be trained for, and whether or not momentum (and if so, which value) or Nesterov acceleration should be used. Take the time to experiment with SGD *as much as you possibly can* and *become comfortable with tuning the parameters*.

Becoming familiar with a given optimization algorithm is similar to mastering how to drive a

car – you drive your own car better than other people's cars because you've spent so much time driving it; you understand your car and its intricacies. Often times, a given optimizer is chosen to train a network on a dataset *not* because the optimizer itself is better, but because the *driver* (i.e., deep learning practitioner) is *more familiar with the optimizer* and understands the "art" behind tuning its respective parameters.

Keep in mind that obtaining a reasonably performing neural network on even a small/medium dataset can take 10's to 100's of experiments even for advanced deep learning users – don't be discouraged when your network isn't performing extremely well right out of the gate. Becoming proficient in deep learning will require an investment of your time and *many* experiments – but it will be worth it once you master how these ingredients come together.

10.1.6 Weight Initialization

Before we close out this chapter I wanted to briefly discuss the concept of *weight initialization*, or more simply, how we initialize our weight matrices and bias vectors.

This section is not meant to be a comprehensive initialization techniques; however, it does highlight popular methods, but from neural network literature and general rules-of-thumb. To illustrate how these weight initialization methods work I have included basic Python/NumPy-like pseudocode when appropriate.

10.1.7 Constant Initialization

When applying constant normalization, all weights in the neural network are initialized with a constant value, C. Typically C will equal zero or one.

To visualize this in pseudocode let's consider an arbitrary layer of a neural network that has 64 inputs and 32 outputs (excluding any biases for notional convenience). To initialize these weights via NumPy and zero initialization (the default used by Caffe, a popular deep learning framework) we would execute:

```
>>> W = np.zeros((64, 32))
```

Similarly, one initialization can be accomplished via:

```
>>> W = np.ones((64, 32))
```

We can apply constant initialization using an arbitrary of C using:

```
>>> W = np.ones((64, 32)) * C
```

Although constant initialization is easy to grasp and understand, the problem with using this method is that its near impossible for us to break the symmetry of activations [114]. Therefore, it is rarely used as a neural network weight initializer.

10.1.8 Uniform and Normal Distributions

A *uniform distribution* draws a random value from the range `[lower, upper]` where every value inside this range has *equal probability* of being drawn.

Again, let's presume that for a given layer in a neural network we have 64 inputs and 32 outputs. We then wish to initialize our weights in the range `lower=-0.05` and `upper=0.05`. Applying the following Python + NumPy code will allow us to achieve the desired normalization:

```
>>> W = np.random.uniform(low=-0.05, high=0.05, size=(64, 32))
```

Executing the code above NumPy will randomly generate $64 \times 32 = 2,048$ values from the range $[-0.05, 0.05]$ where each value in this range has equal probability.

We then have a *normal distribution* where we define the probability density for the Gaussian distribution as:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{10.6}$$

The most important parameters here are μ (the mean) and σ (the standard deviation). The square of the standard deviation, σ^2, is called the variance.

When using the Keras library the `RandomNormal` class draws random values from a normal distribution with $\mu = 0$ and $\sigma = 0.05$. We can mimic this behavior using NumPy below:

```
>>> W = np.random.normal(0.0, 0.5, size=(64, 32))
```

Both of uniform and normal distributions can be used to initialize the weights in neural networks; however, we normally impose various heuristics to create "better" initialization schemes (as we'll discuss in the remaining sections).

10.1.9 LeCun Uniform and Normal

If you have ever used the Torch7 or PyTorch frameworks you may notice that the default weight initialization method is called "Efficient Backprop", which is derived by the work of LeCun et al. [17].

Here the authors define a parameter F_{in} (called "fan in", or the number of *inputs* to the layer) along with F_{out} (the "fan out", or number of *outputs* from the layer). Using these values we can apply uniform initialization by:

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(3 / float(F_in))
>>> W = np.random.uniform(low=-limit, high=limit, size=(F_in, F_out))
```

We can also use a normal distribution as well. The Keras library uses a truncated normal distribution when constructing the lower and upper limits, along with a zero mean:

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(1 / float(F_in))
>>> W = np.random.normal(0.0, limit, size=(F_in, F_out))
```

10.1.10 Glorot/Xavier Uniform and Normal

The default weight initialization method used in the Keras library is called "Glorot initialization" or "Xavier initialization" named after Xavier Glorot, the first author of the paper, *Understanding the difficulty of training deep feedforward neural networks* [115].

For the normal distribution the `limit` value is constructed by *averaging* the F_{in} and F_{out} together and then taking the square-root [116]. A zero-center ($\mu = 0$) is then used:

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(2 / float(F_in + F_out))
>>> W = np.random.normal(0.0, limit, size=(F_in, F_out))
```

Glorot/Xavier initialization can also be done with a uniform distribution where we place stronger restrictions on `limit`:

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(6 / float(F_in + F_out))
>>> W = np.random.uniform(low=-limit, high=limit, size=(F_in, F_out))
```

Learning tends to be quite efficient using this initialization method and I recommend it for most neural networks.

10.1.11 He et al./Kaiming/MSRA Uniform and Normal

Often referred to as "He et al. initialization", "Kaiming initialization", or simply "MSRA initialization", this technique is named after Kaiming He, the first author of the paper, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification* [117].

We typically used this method when we are training *very deep* neural networks that use a ReLU-like activation function (in particular, a "PReLU", or Parametric Rectified Linear Unit).

To initialize the weights in a layer using He et al. initialization with a *uniform distribution* we set `limit` to be $limit = \sqrt{6/F_{in}}$, where F_{in} is the number of input units in the layer:

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(6 / float(F_in))
>>> W = np.random.uniform(low=-limit, high=limit, size=(F_in, F_out))
```

We can also use a *normal distribution* as well by setting $\mu = 0$ and $sigma = \sqrt{2/F_{in}}$

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(2 / float(F_in))
>>> W = np.random.normal(0.0, limit, size=(F_in, F_out))
```

We'll discuss this initialization method in both the *Practitioner Bundle* and the *ImageNet Bundle* of this book where we train very deep neural networks on large image datasets.

10.1.12 Differences in Initialization Implementation

The actual `limit` values may vary for LeCun Uniform/Normal, Xavier Uniform/Normal, and He et al. Uniform/Normal. For example, when using Xavier Uniform in Caffe, `limit = -np.sqrt(3 / F_in)` [114]; however, the default Xaiver initialization for Keras uses `np.sqrt(6 / (F_in + F_out))` [118]. No method is "more correct" than the other, but you should read the documentation of your respective deep learning library.

10.2 Summary

In this chapter, we reviewed the fundamentals of neural networks. Specifically, we focused on the history of neural networks and the relation to biology.

From there, we moved on to *artificial neural networks*, such as the Perceptron algorithm. While important from a historical standpoint, the Perceptron algorithm has one major flaw – it accurately classifies nonlinear separable points. In order to work with more challenging datasets we need both (1) nonlinear activation functions and (2) multi-layer networks.

To train multi-layer networks we must use the backpropagation algorithm. We then implemented backpropagation by hand and demonstrated that when used to train multi-layer networks with nonlinear activation functions, we can model nonlinearly separable datasets, such as XOR.

Of course, implementing backpropagation by hand is an arduous process prone to bugs – we, therefore, often rely on existing libraries such as Keras, Theano, TensorFlow, etc. This enables us to focus on the actual *architecture* rather than the underlying algorithm used to train the network.

Finally, we reviewed the four key ingredients when working with *any* neural network, including the *dataset*, *loss function*, *model/architecture*, and *optimization method*.

Unfortunately, as some of our results demonstrated (e.x., CIFAR-10) standard neural networks fail to obtain high classification accuracy when working with challenging image datasets that exhibit variations in translation, rotation, viewpoint, etc. In order to obtain reasonable accuracy on these datasets, we'll need to work with a special type of feedforward neural networks called *Convolutional Neural Networks* (CNNs), which is exactly the subject of our next chapter.