O'REILLY®

Deep Learning from Scratch

Building with Python from First Principles

Seth Weidman

Deep Learning from Scratch

Building with Python from First Principles

Seth Weidman

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

Deep Learning from Scratch

by Seth Weidman

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

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

Development Editor: Melissa Potter

Acquisitions Editors: Jon Hassell and Mike Loukides

Production Editor: Katherine Tozer

Copyeditor: Arthur Johnson

Proofreader: Rachel Monaghan

Indexer: Judith McConville

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

September 2019: First Edition

Revision History for the First Edition

- 2019-09-06: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492041412 for release

978-1-492-04141-2

[LSI]

Preface

If you’ve tried to learn about neural networks and deep learning, you’ve probably encountered an abundance of resources, from blog posts to MOOCs (massive open online courses, such as those offered on Coursera and Udacity) of varying quality and even some books—I know I did when I started exploring the subject a few years ago. However, if you’re reading this preface, it’s likely that each explanation of neural networks that you’ve come across is lacking in some way. I found the same thing when I started learning: the various explanations were like blind men describing different parts of an elephant, but none describing the whole thing. That is what led me to write this book.

These existing resources on neural networks mostly fall into two categories. Some are conceptual and mathematical, containing both the drawings one typically finds in explanations of neural networks, of circles connected by lines with arrows on the ends, as well as extensive mathematical explanations of what is going on so you can “understand the theory.” A prototypical example of this is the very good book *Deep Learning* by Ian Goodfellow et al. (MIT Press).

Other resources have dense blocks of code that, if run, appear to show a loss value decreasing over time and thus a neural network “learning.” For instance, the following example from the PyTorch documentation does indeed define and train a simple neural network on randomly generated data:

```
# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = torch.randn(N, D_in, device=device, dtype=dtype)
y = torch.randn(N, D_out, device=device, dtype=dtype)

# Randomly initialize weights
w1 = torch.randn(D_in, H, device=device, dtype=dtype)
w2 = torch.randn(H, D_out, device=device, dtype=dtype)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
```

```
h = x.mm(w1)
h_relu = h.clamp(min=0)
y_pred = h_relu.mm(w2)

# Compute and print loss
loss = (y_pred - y).pow(2).sum().item()
print(t, loss)

# Backprop to compute gradients of w1 and w2 with respect to loss
grad_y_pred = 2.0 * (y_pred - y)
grad_w2 = h_relu.t().mm(grad_y_pred)
grad_h_relu = grad_y_pred.mm(w2.t())
grad_h = grad_h_relu.clone()
grad_h[h < 0] = 0
grad_w1 = x.t().mm(grad_h)

# Update weights using gradient descent
w1 -= learning_rate * grad_w1
w2 -= learning_rate * grad_w2
```

Explanations like this, of course, don't give much insight into "what is really going on": the underlying mathematical principles, the individual neural network components contained here and how they work together, and so on.[1]

What *would* a good explanation of neural networks contain? For an answer, it is instructive to look at how other computer science concepts are explained: if you want to learn about sorting algorithms, for example, there are textbooks that will contain:

- An explanation of the algorithm, in plain English
- A visual explanation of how the algorithm works, of the kind that you would draw on a whiteboard during a coding interview
- Some mathematical explanation of "why the algorithm works"[2]
- Pseudocode implementing the algorithm

One rarely—or never—finds these elements of an explanation of neural networks side by side, even though it seems obvious to me that a proper explanation of neural networks should be done this way; this book is an attempt to fill that gap.

Understanding Neural Networks Requires Multiple Mental Models

I am not a researcher, and I do not have a Ph.D. I have, however, taught data science professionally: I taught a couple of data science bootcamps with a company called Metis, and then I traveled around the world for a year with Metis doing one- to five-day workshops for companies in many different industries in which I explained machine learning and basic software engineering concepts to their employees. I've always loved teaching and have always been fascinated by the question of how best to explain technical concepts, most recently focusing on concepts in machine learning and statistics. With neural networks, I've found the most challenging part is conveying the correct "mental model" for what a neural network is, especially since understanding neural networks fully requires not just one but *several* mental models, all of which illuminate different (but still essential) aspects of how neural networks work. To illustrate this: the following four sentences are all correct answers to the question "What is a neural network?":

- A neural network is a mathematical function that takes in inputs and produces outputs.
- A neural network is a computational graph through which multidimensional arrays flow.
- A neural network is made up of layers, each of which can be thought of as having a number of "neurons."
- A neural network is a universal function approximator that can in theory represent the solution to any supervised learning problem.

Indeed, many of you reading this have probably heard one or more of these before, and may have a reasonable understanding of what they mean and what their implications are for how neural networks work. To fully understand them, however, we'll have to understand *all* of them and show how they are connected —how is the fact that a neural network can be represented as a computational graph connected to the notion of "layers," for example? Furthermore, to make all of this precise, we'll implement all of these concepts from scratch, in Python, and stitch them together to make working neural networks that you can train on your laptop. Nevertheless, despite the fact that we'll spend a substantial amount of time on implementation details, *the purpose of implementing these models in Python is to solidify and make precise our understanding of the concepts; it is not to write as concise or performant of a neural network library as possible.*

My goal is that after you've read this book, you'll have such a solid understanding of all of these mental models (and their implications for how neural networks should be *implemented*) that learning related concepts or doing further projects in the field will be much easier.

Chapter Outlines

The first three chapters are the most important ones and could themselves form a standalone book.

1. In Chapter 1 I'll show how mathematical functions can be represented as a series of operations linked together to form a computational graph, and show how this representation lets us compute the derivatives of these functions' outputs with respect to their inputs using the chain rule from calculus. At the end of this chapter, I'll introduce a very important operation, the matrix multiplication, and show how it can fit into a mathematical function represented in this way while still allowing us to compute the derivatives we'll end up needing for deep learning.

2. In Chapter 2 we'll directly use the building blocks we created in Chapter 1 to build and train models to solve a real-world problem: specifically, we'll use them to build both linear regression and neural network models to predict housing prices on a real-world dataset. I'll show that the neural network performs better than the linear regression and try to give some intuition for why. The "first principles" approach to building the models in this chapter should give you a very good idea of how neural networks work, but will also show the limited capability of the step-by-step, purely first-principles-based approach to defining deep learning models; this will motivate Chapter 3.

3. In Chapter 3 we'll take the building blocks from the first-principles-based approach of the first two chapters and use them to build the "higher level" components that make up all deep learning models: `Layers`, `Models`, `Optimizers`, and so on. We'll end this chapter by training a deep learning model, defined from scratch, on the same dataset from Chapter 2 and showing that it performs better than our simple neural network.

4. As it turns out, there are few theoretical guarantees that a neural network with a given architecture will actually find a good solution on a given dataset when trained using the standard training techniques we'll

use in this book. In Chapter 4 we’ll cover the most important “training tricks” that generally increase the probability that a neural network will find a good solution, and, wherever possible, give some mathematical intuition as to why they work.

5. In Chapter 5 I cover the fundamental ideas behind convolutional neural networks (CNNs), a kind of neural network architecture specialized for understanding images. There are many explanations of CNNs out there, so I’ll focus on explaining the absolute essentials of CNNs and how they differ from regular neural networks: specifically, how CNNs result in each layer of neurons being organized into “feature maps,” and how two of these layers (each made up of multiple feature maps) are connected together via convolutional filters. In addition, just as we coded the regular layers in a neural network from scratch, we’ll code convolutional layers from scratch to reinforce our understanding of how they work.

6. Throughout the first five chapters, we’ll build up a miniature neural network library that defines neural networks as a series of `Layers`—which are themselves made up of a series of `Operations`—that send inputs forward and gradients backward. This is not how most neural networks are implemented in practice; instead, they use a technique called *automatic differentiation*. I’ll give a quick illustration of automatic differentiation at the beginning of Chapter 6 and use it to motivate the main subject of the chapter: *recurrent neural networks* (RNNs), the neural network architecture typically used for understanding data in which the data points appear sequentially, such as time series data or natural language data. I’ll explain the workings of “vanilla RNNs” and of two variants: *GRUs* and *LSTMs* (and of course implement all three from scratch); throughout, I’ll be careful to distinguish between the elements that are shared across *all* of these RNN variants and the specific ways in which these variants differ.

7. Finally, in Chapter 7, I’ll show how everything we did from scratch in Chapters 1–6 can be implemented using the high-performance, open source neural network library PyTorch. Learning a framework like this is essential for progressing your learning about neural networks; but

> diving in and learning a framework without first having a solid understanding of how and why neural networks work would severely limit your learning in the long term. The goal of the progression of chapters in this book is to give you the power to write extremely high-performance neural networks (by teaching you PyTorch) while still setting you up for long-term learning and success (by teaching you the fundamentals before you learn PyTorch). We'll conclude with a quick illustration of how neural networks can be used for unsupervised learning.

My goal here was to write the book that I wish had existed when I started to learn the subject a few years ago. I hope you will find this book helpful. Onward!

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

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

`Constant width`

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

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Used for text that should be replaced with user-supplied values or by values determined by context and for comments in code examples.

The Pythagorean Theorem is $a^2 + b^2 = c^2$.

NOTE

This element signifies a general note.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at the book’s GitHub repository.

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

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Deep Learning from Scratch* by Seth Weidman (O’Reilly). Copyright 2019 Seth Weidman, 978-1-492-04141-2.”

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

O’Reilly Online Learning

NOTE

For almost 40 years, *O’Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O’Reilly’s online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

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

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

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/dl-from-scratch*.

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

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

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

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

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

Acknowledgments

I'd like to thank my editor, Melissa Potter, along with the team at O'Reilly, who were meticulous with their feedback and responsive to my questions throughout the process.

I’d like to give a special thanks to several people whose work to make technical concepts in machine learning accessible to a wider audience has directly influenced me, and a couple of whom I’ve been lucky enough to have gotten to know personally: in a randomly generated order, these people are Brandon Rohrer, Joel Grus, Jeremy Watt, and Andrew Trask.

I’d like to thank my boss at Metis and my director at Facebook, who were unreasonably supportive of my carving out time to work on this project.

I’d like to give a special thank you and acknowledgment to Mat Leonard, who was my coauthor for a brief period of time before we decided to go our separate ways. Mat helped organize the code for the minilibrary associated with the book —`lincoln`—and gave me very helpful feedback on some extremely unpolished versions of the first two chapters, writing his own versions of large sections of these chapters in the process.

Finally, I’d like to thank my friends Eva and John, both of whom directly encouraged and inspired me to take the plunge and actually start writing. I’d also like to thank my many friends in San Francisco who tolerated my general preoccupation and worry about the book as well as my lack of availability to hang out for many months, and who were unwaveringly supportive when I needed them to be.

---

1 To be fair, this example was intended as an illustration of the PyTorch library for those who already understand neural networks, not as an instructive tutorial. Still, many tutorials follow this style, showing only the code along with some brief explanations.

2 Specifically, in the case of sorting algorithms, why the algorithm terminates with a properly sorted list.

Chapter 1. Foundations

Don't memorize these formulas. If you understand the concepts, you can invent your own notation.

—John Cochrane, *Investments Notes* 2006

The aim of this chapter is to explain some foundational mental models that are essential for understanding how neural networks work. Specifically, we'll cover *nested mathematical functions and their derivatives*. We'll work our way up from the simplest possible building blocks to show that we can build complicated functions made up of a "chain" of constituent functions and, even when one of these functions is a matrix multiplication that takes in multiple inputs, compute the derivative of the functions' outputs with respect to their inputs. Understanding how this process works will be essential to understanding neural networks, which we technically won't begin to cover until Chapter 2.

As we're getting our bearings around these foundational building blocks of neural networks, we'll systematically describe each concept we introduce from three perspectives:

- Math, in the form of an equation or equations
- Code, with as little extra syntax as possible (making Python an ideal choice)
- A diagram explaining what is going on, of the kind you would draw on a whiteboard during a coding interview

As mentioned in the preface, one of the challenges of understanding neural networks is that it requires multiple mental models. We'll get a sense of that in this chapter: each of these three perspectives excludes certain essential features of the concepts we'll cover, and only when taken together do they provide a full picture of both how and why nested mathematical functions work the way they do. In fact, I take the uniquely strong view that any attempt to explain the building blocks of neural networks that excludes one of these three perspectives is incomplete.

With that out of the way, it's time to take our first steps. We're going to start with some extremely simple building blocks to illustrate how we can understand different concepts in terms of these three perspectives. Our first building block will be a simple but critical concept: the function.

Functions

What is a function, and how do we describe it? As with neural nets, there are several ways to describe functions, none of which individually paints a complete picture. Rather than trying to give a pithy one-sentence description, let's simply walk through the three mental models one by one, playing the role of the blind men feeling different parts of the elephant.

Math

Here are two examples of functions, described in mathematical notation:

- $f_1(x) = x^2$
- $f_2(x) = max(x, 0)$

This notation says that the functions, which we arbitrarily call f_1 and f_2, take in a number x as input and transform it into either x^2 (in the first case) or $max(x, 0)$ (in the second case).

Diagrams

One way of depicting functions is to:

1. Draw an x-y plane (where x refers to the horizontal axis and y refers to the vertical axis).
2. Plot a bunch of points, where the x-coordinates of the points are (usually evenly spaced) inputs of the function over some range, and the y-coordinates are the outputs of the function over that range.
3. Connect these plotted points.

This was first done by the French philosopher René Descartes, and it is extremely useful in many areas of mathematics, in particular calculus. Figure 1-1 shows the plot of these two functions.

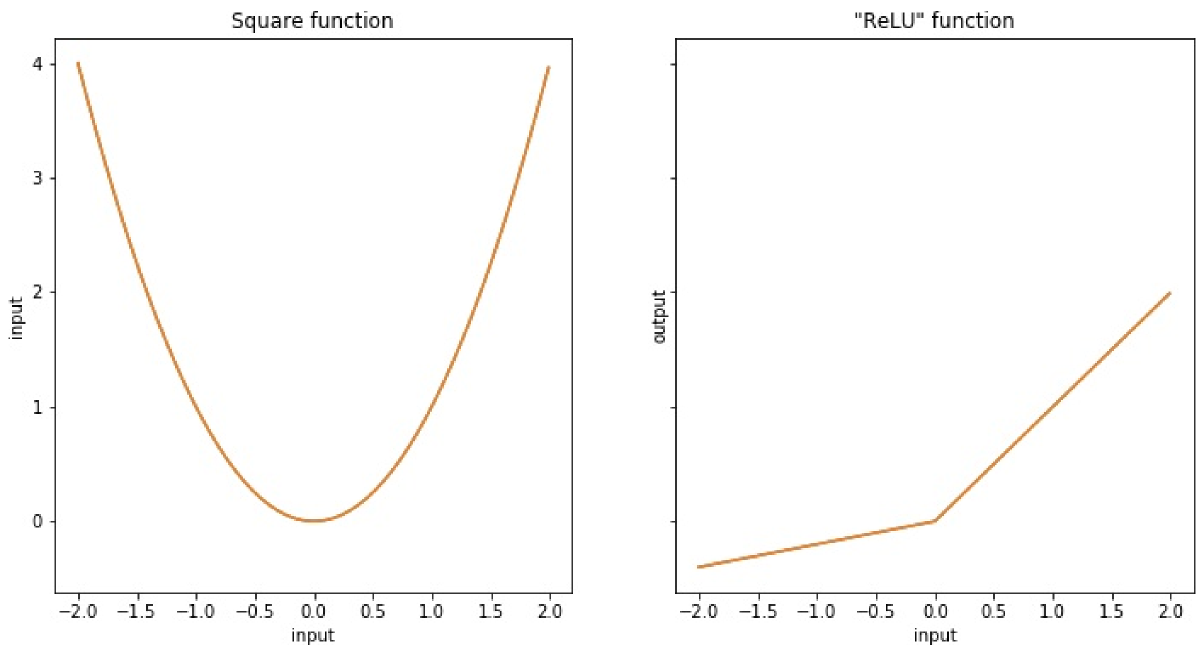


Figure 1-1. Two continuous, mostly differentiable functions

However, there is another way to depict functions that isn't as useful when learning calculus but that will be very useful for us when thinking about deep learning models. We can think of functions as boxes that take in numbers as input and produce numbers as output, like minifactories that have their own internal rules for what happens to the input. Figure 1-2 shows both these functions described as general rules and how they operate on specific inputs.

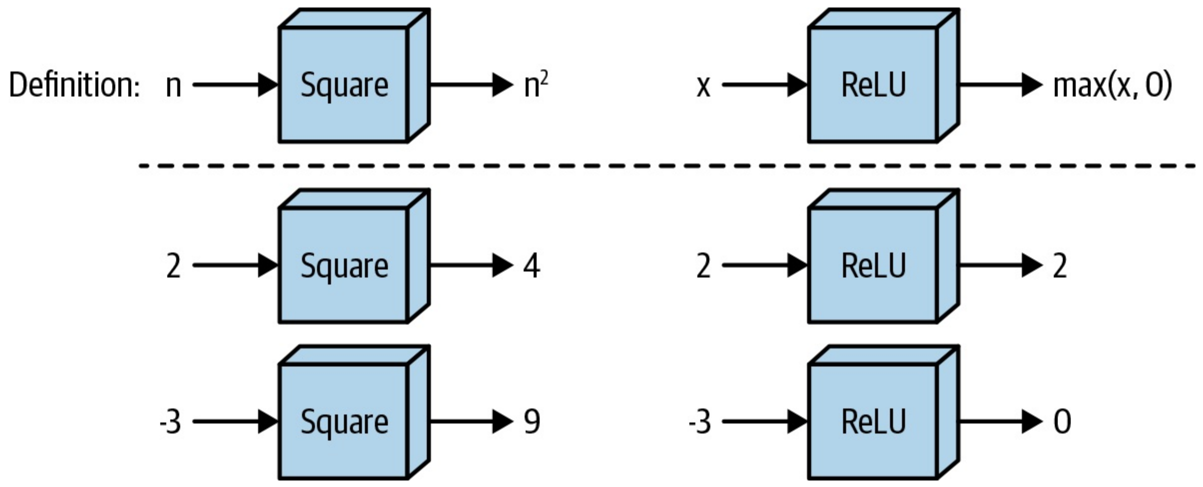


Figure 1-2. Another way of looking at these functions

Code

Finally, we can describe these functions using code. Before we do, we should say a bit about the Python library on top of which we'll be writing our functions: NumPy.

Code caveat #1: NumPy

NumPy is a widely used Python library for fast numeric computation, the internals of which are mostly written in C. Simply put: the data we deal with in neural networks will always be held in a *multidimensional array* that is almost always either one-, two-, three-, or four-dimensional, but especially two- or three-dimensional. The `ndarray` class from the NumPy library allows us to operate on these arrays in ways that are both (a) intuitive and (b) fast. To take the simplest possible example: if we were storing our data in Python lists (or lists of lists), adding or multiplying the lists elementwise using normal syntax wouldn't work, whereas it does work for `ndarrays`:

```
print("Python list operations:")
a = [1,2,3]
b = [4,5,6]
print("a+b:", a+b)
try:
    print(a*b)
except TypeError:
    print("a*b has no meaning for Python lists")
print()
print("numpy array operations:")
a = np.array([1,2,3])
b = np.array([4,5,6])
print("a+b:", a+b)
print("a*b:", a*b)
```

```
Python list operations:
a+b: [1, 2, 3, 4, 5, 6]
a*b has no meaning for Python lists

numpy array operations:
a+b: [5 7 9]
a*b: [ 4 10 18]
```

`ndarrays` also have several features you'd expect from an `n`-dimensional array;

each `ndarray` has `n` axes, indexed from 0, so that the first axis is `0`, the second is `1`, and so on. In particular, since we deal with 2D `ndarrays` often, we can think of `axis = 0` as the rows and `axis = 1` as the columns—see Figure 1-3.

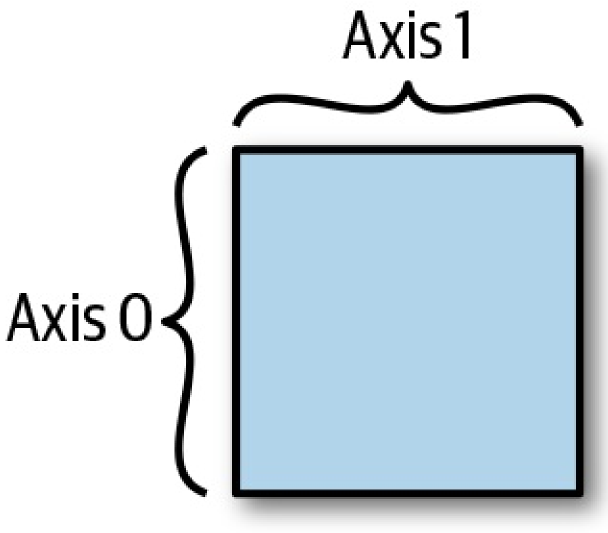


Figure 1-3. A 2D NumPy array, with axis = 0 as the rows and axis = 1 as the columns

NumPy's `ndarrays` also support applying functions along these axes in intuitive ways. For example, summing along axis 0 (the *rows* for a 2D array) essentially "collapses the array" along that axis, returning an array with one less dimension than the original array; for a 2D array, this is equivalent to summing each column:

```
print('a:')
print(a)
print('a.sum(axis=0):', a.sum(axis=0))
print('a.sum(axis=1):', a.sum(axis=1))
```

```
a:
[[1 2]
 [3 4]]
a.sum(axis=0): [4 6]
a.sum(axis=1): [3 7]
```

Finally, NumPy `ndarrays` support adding a 1D array to the last axis; for a 2D array `a` with `R` rows and `C` columns, this means we can add a 1D array `b` of length `C` and NumPy will do the addition in the intuitive way, adding the elements to each row of `a`:[1]

```
a = np.array([[1,2,3],
              [4,5,6]])

b = np.array([10,20,30])

print("a+b:\n", a+b)
```

```
a+b:
[[11 22 33]
 [14 25 36]]
```

Code caveat #2: Type-checked functions

As I've mentioned, the primary goal of the code we write in this book is to make the concepts I'm explaining precise and clear. This will get more challenging as the book goes on, as we'll be writing functions with many arguments as part of complicated classes. To combat this, we'll use functions with type signatures throughout; for example, in Chapter 3, we'll initialize our neural networks as follows:

```
def __init__(self,
             layers: List[Layer],
             loss: Loss,
             learning_rate: float = 0.01) -> None:
```

This type signature alone gives you some idea of what the class is used for. By contrast, consider the following type signature that we *could* use to define an operation:

```
def operation(x1, x2):
```

This type signature by itself gives you no hint as to what is going on; only by printing out each object's type, seeing what operations get performed on each object, or guessing based on the names `x1` and `x2` could we understand what is going on in this function. I can instead define a function with a type signature as follows:

```
def operation(x1: ndarray, x2: ndarray) -> ndarray:
```

You know right away that this is a function that takes in two `ndarrays`, probably combines them in some way, and outputs the result of that combination. Because of the increased clarity they provide, we'll use type-checked functions throughout this book.

Basic functions in NumPy

With these preliminaries in mind, let's write up the functions we defined earlier in NumPy:

```python
def square(x: ndarray) -> ndarray:
    '''
    Square each element in the input ndarray.
    '''
    return np.power(x, 2)

def leaky_relu(x: ndarray) -> ndarray:
    '''
    Apply "Leaky ReLU" function to each element in ndarray.
    '''
    return np.maximum(0.2 * x, x)
```

NOTE

One of NumPy's quirks is that many functions can be applied to `ndarray`s either by writing `np.`*`function_name`*`(ndarray)` or by writing `ndarray.`*`function_name`*. For example, the preceding `relu` function could be written as: `x.clip(min=0)`. We'll try to be consistent and use the `np.`*`function_name`*`(ndarray)` convention throughout—in particular, we'll avoid tricks such as *`ndarray`*`.T` for transposing a two-dimensional `ndarray`, instead writing `np.transpose(`*`ndarray`*`, (1, 0))`.

If you can wrap your mind around the fact that math, a diagram, and code are three different ways of representing the same underlying concept, then you are well on your way to displaying the kind of flexible thinking you'll need to truly understand deep learning.

Derivatives

Derivatives, like functions, are an extremely important concept for understanding deep learning that many of you are probably familiar with. Also like functions, they can be depicted in multiple ways. We’ll start by simply saying at a high level that the derivative of a function at a point is the “rate of change” of the output of the function with respect to its input at that point. Let’s now walk through the same three perspectives on derivatives that we covered for functions to gain a better mental model for how derivatives work.

Math

First, we’ll get mathematically precise: we can describe this number—how much the output of *f* changes as we change its input at a particular value *a* of the input —as a limit:

$$\frac{df}{du}(a) = \lim_{\Delta \to 0} \frac{f(a + \Delta) - f(a - \Delta)}{2 \times \Delta}$$

This limit can be approximated numerically by setting a very small value for *Δ*, such as 0.001, so we can compute the derivative as:

$$\frac{df}{du}(a) = \frac{f(a + 0.001) - f(a - 0.001)}{0.002}$$

While accurate, this is only one part of a full mental model of derivatives. Let’s look at them from another perspective: a diagram.

Diagrams

First, the familiar way: if we simply draw a tangent line to the Cartesian representation of the function *f*, the derivative of *f* at a point *a* is just the slope of this line at *a*. As with the mathematical descriptions in the prior subsection, there are two ways we can actually calculate the slope of this line. The first would be to use calculus to actually calculate the limit. The second would be to just take the slope of the line connecting *f* at *a* – 0.001 and *a* + 0.001. The latter method is depicted in Figure 1-4 and should be familiar to anyone who has taken calculus.

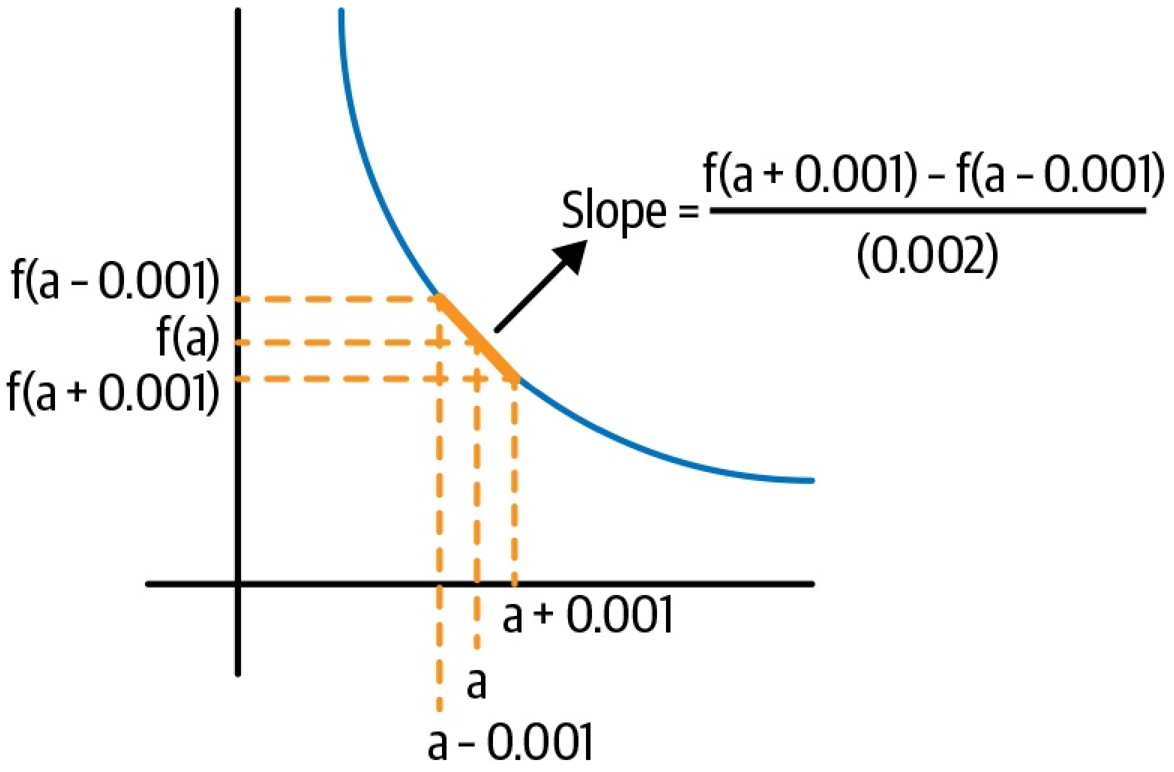


Figure 1-4. Derivatives as slopes

As we saw in the prior section, another way of thinking of functions is as mini-factories. Now think of the inputs to those factories being connected to the outputs by a string. The derivative is equal to the answer to this question: if we pull up on the input to the function *a* by some very small amount—or, to account for the fact that the function may be asymmetric at *a*, pull down on *a* by some small amount—by what multiple of this small amount will the output change, given the inner workings of the factory? This is depicted in Figure 1-5.

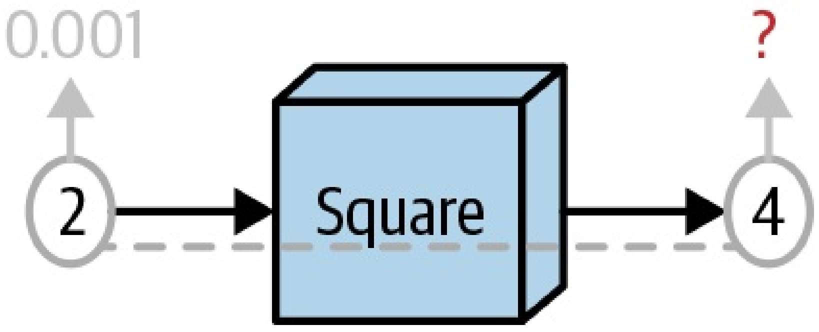


Figure 1-5. Another way of visualizing derivatives

This second representation will turn out to be more important than the first one for understanding deep learning.

Code

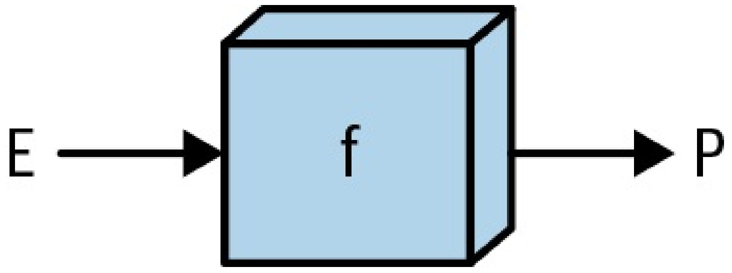
Finally, we can code up the approximation to the derivative that we saw previously:

```
from typing import Callable

def deriv(func: Callable[[ndarray], ndarray],
          input_: ndarray,
          delta: float = 0.001) -> ndarray:
    '''
    Evaluates the derivative of a function "func" at every element in the
    "input_" array.
    '''
    return (func(input_ + delta) - func(input_ - delta)) / (2 * delta)
```

NOTE

When we say that "something is a function of something else"—for example, that *P* is a function of *E* (letters chosen randomly on purpose), what we mean is that there is some function *f* such that *f*(*E*) = *P*—or equivalently, there is a function *f* that takes in *E* objects and produces *P* objects. We might also think of this as meaning that *P is defined as* whatever results when we apply the function *f* to *E*:



And we would code this up as:

```
def f(input_: ndarray) -> ndarray:
    # Some transformation(s)
    return output

P = f(E)
```

Nested Functions

Now we'll cover a concept that will turn out to be fundamental to understanding neural networks: functions can be "nested" to form "composite" functions. What exactly do I mean by "nested"? I mean that if we have two functions that by mathematical convention we call f_1 and f_2, the output of one of the functions becomes the input to the next one, so that we can "string them together."

Diagram

The most natural way to represent a nested function is with the "minifactory" or "box" representation (the second representation from "Functions").

As Figure 1-6 shows, an input goes into the first function, gets transformed, and comes out; then it goes into the second function and gets transformed again, and we get our final output.

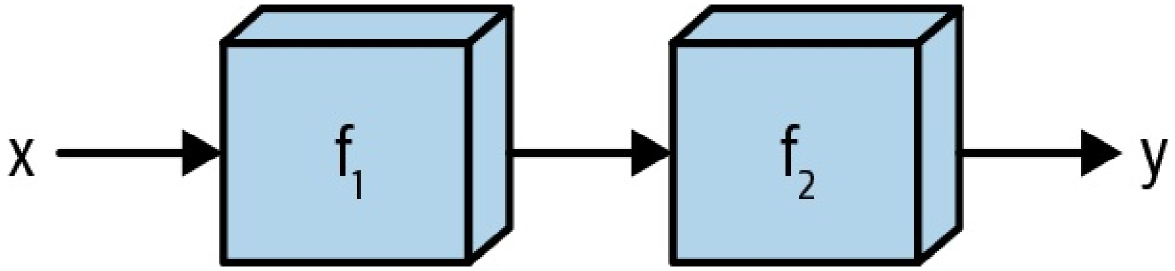


Figure 1-6. Nested functions, naturally

Math

We should also include the less intuitive mathematical representation:

$$f_2\left(f_1\left(x\right)\right) = y$$

This is less intuitive because of the quirk that nested functions are read "from the outside in" but the operations are in fact performed "from the inside out." For example, though $f_2\left(f_1\left(x\right)\right) = y$ is read "f 2 of f 1 of x," what it really means is to "first apply f_1 to x, and then apply f_2 to the result of applying f_1 to x."

Code

Finally, in keeping with my promise to explain every concept from three perspectives, we'll code this up. First, we'll define a data type for nested functions:

```
from typing import List

# A Function takes in an ndarray as an argument and produces an ndarray
Array_Function = Callable[[ndarray], ndarray]

# A Chain is a list of functions
Chain = List[Array_Function]
```

Then we'll define how data goes through a chain, first of length 2:

```
def chain_length_2(chain: Chain,
                   a: ndarray) -> ndarray:
    '''
    Evaluates two functions in a row, in a "Chain".
    '''
    assert len(chain) == 2, \
    "Length of input 'chain' should be 2"

    f1 = chain[0]
    f2 = chain[1]

    return f2(f1(x))
```

Another Diagram

Depicting the nested function using the box representation shows us that this composite function is really just a single function. Thus, we can represent this function as simply $f_1 f_2$, as shown in Figure 1-7.

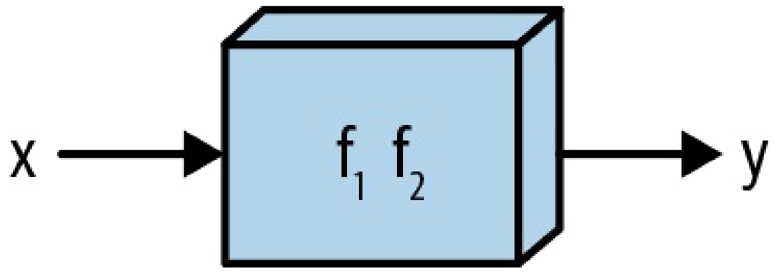


Figure 1-7. Another way to think of nested functions

Moreover, a theorem from calculus tells us that a composite function made up of "mostly differentiable" functions is itself mostly differentiable! Thus, we can think of $f_1 f_2$ as just another function that we can compute derivatives of—and computing derivatives of composite functions will turn out to be essential for training deep learning models.

However, we need a formula to be able to compute this composite function's derivative in terms of the derivatives of its constituent functions. That's what we'll cover next.

The Chain Rule

The chain rule is a mathematical theorem that lets us compute derivatives of composite functions. Deep learning models are, mathematically, composite functions, and reasoning about their derivatives is essential to training them, as we'll see in the next couple of chapters.

Math

Mathematically, the theorem states—in a rather nonintuitive form—that, for a given value `x`,

$$\frac{df_2}{du}(x) = \frac{df_2}{du}(f_1(x)) \times \frac{df_1}{du}(x)$$

where *u* is simply a dummy variable representing the input to a function.

NOTE

When describing the derivative of a function *f* with one input and output, we can denote the *function* that represents the derivative of this function as $\frac{df}{du}$. We could use a different dummy variable in place of *u*—it doesn't matter, just as $f(x) = x^2$ and $f(y) = y^2$ mean the same thing.

On the other hand, later on we'll deal with functions that take in *multiple* inputs, say, both *x* and *y*. Once we get there, it will make sense to write $\frac{df}{dx}$ and have it mean something different than $\frac{df}{dy}$.

This is why in the preceding formula we denote *all* the derivatives with a *u* on the bottom: both f_1 and f_2 are functions that take in one input and produce one output, and in such cases (of functions with one input and one output) we'll use *u* in the derivative notation.

Diagram

The preceding formula does not give much intuition into the chain rule. For that, the box representation is much more helpful. Let's reason through what the derivative "should" be in the simple case of $f_1 f_2$.

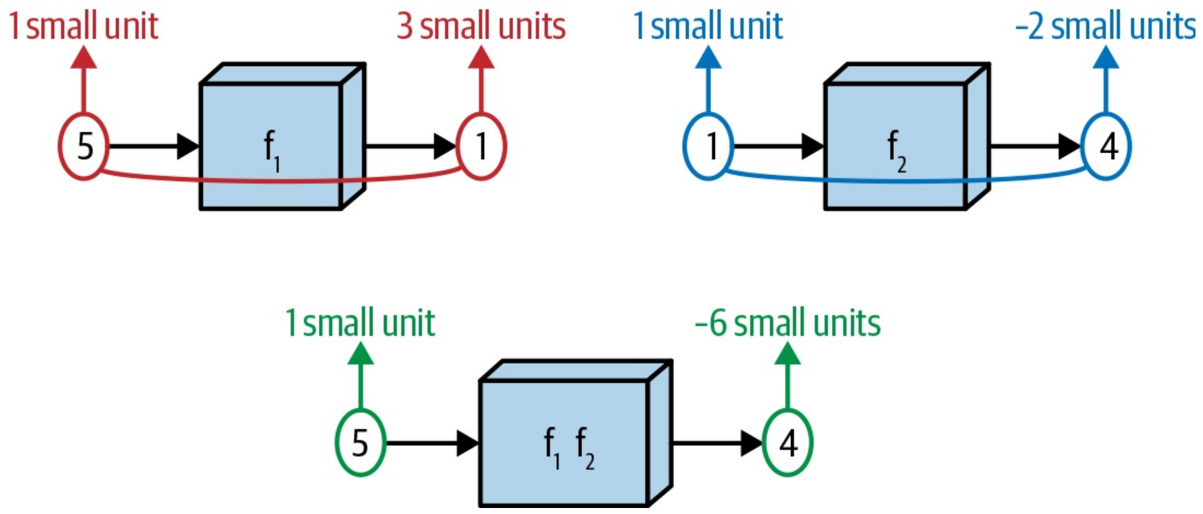


Figure 1-8. An illustration of the chain rule

Intuitively, using the diagram in Figure 1-8, the derivative of the composite function *should* be a sort of product of the derivatives of its constituent functions. Let's say we feed the value 5 into the first function, and let's say further that computing the *derivative* of the first function at $u = 5$ gives us a value of 3—that is, $\frac{df_1}{du}(5) = 3$.

Let's say that we then take the *value* of the function that comes out of the first box—let's suppose it is 1, so that $f_1(5) = 1$—and compute the derivative of the second function f_2 at this value: that is, $\frac{df_2}{du}(1)$. We find that this value is –2.

If we think about these functions as being literally strung together, then if changing the input to box two by 1 unit yields a change of –2 units in the output of box two, changing the input to box two by 3 units should change the output to box two by $-2 \times 3 = -6$ units. This is why in the formula for the chain rule, the final result is ultimately a product: $\frac{df_2}{du}(f_1(x))$ *times* $\frac{df_1}{du}(x)$.

So by considering the diagram and the math, we can reason through what the derivative of the output of a nested function with respect to its input ought to be, using the chain rule. What might the code instructions for the computation of this derivative look like?

Code

Let’s code this up and show that computing derivatives in this way does in fact yield results that “look correct.” We’ll use the `square` function from “Basic functions in NumPy” along with `sigmoid`, another function that ends up being important in deep learning:

```
def sigmoid(x: ndarray) -> ndarray:
    '''
    Apply the sigmoid function to each element in the input ndarray.
    '''
    return 1 / (1 + np.exp(-x))
```

And now we code up the chain rule:

```
def chain_deriv_2(chain: Chain,
                  input_range: ndarray) -> ndarray:
    '''
    Uses the chain rule to compute the derivative of two nested functions:
    (f2(f1(x))' = f2'(f1(x)) * f1'(x)
    '''

    assert len(chain) == 2, \
    "This function requires 'Chain' objects of length 2"

    assert input_range.ndim == 1, \
    "Function requires a 1 dimensional ndarray as input_range"

    f1 = chain[0]
    f2 = chain[1]

    # df1/dx
    f1_of_x = f1(input_range)

    # df1/du
    df1dx = deriv(f1, input_range)

    # df2/du(f1(x))
    df2du = deriv(f2, f1(input_range))

    # Multiplying these quantities together at each point
    return df1dx * df2du
```

Figure 1-9 plots the results and shows that the chain rule works:

```
PLOT_RANGE = np.arange(-3, 3, 0.01)

chain_1 = [square, sigmoid]
chain_2 = [sigmoid, square]

plot_chain(chain_1, PLOT_RANGE)
plot_chain_deriv(chain_1, PLOT_RANGE)

plot_chain(chain_2, PLOT_RANGE)
plot_chain_deriv(chain_2, PLOT_RANGE)
```

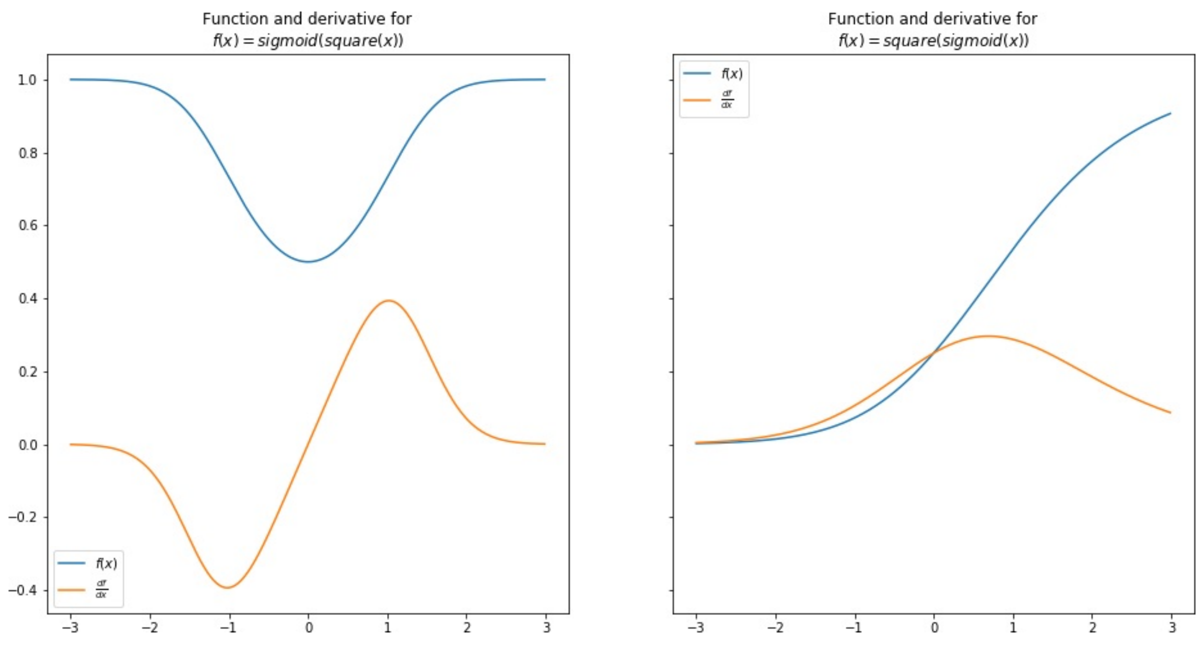


Figure 1-9. The chain rule works, part 1

The chain rule seems to be working. When the functions are upward-sloping, the derivative is positive; when they are flat, the derivative is zero; and when they are downward-sloping, the derivative is negative.

So we can in fact compute, both mathematically and via code, the derivatives of nested or "composite" functions such as $f_1 f_2$, as long as the individual functions are themselves mostly differentiable.

It will turn out that deep learning models are, mathematically, long chains of these mostly differentiable functions; spending time going manually through a slightly longer example in detail will help build your intuition about what is going on and how it can generalize to more complex models.

A Slightly Longer Example

Let's closely examine a slightly longer chain: if we have three mostly differentiable functions—f_1, f_2, and f_3—how would we go about computing the derivative of f_1 f_2 f_3? We "should" be able to do it, since from the calculus theorem mentioned previously, we know that the composite of *any* finite number of "mostly differentiable" functions is differentiable.

Math

Mathematically, the result turns out to be the following expression:

$$\frac{df_3}{du}(x) = \frac{df_3}{du}(f_2(f_1(x))) \times \frac{df_2}{du}(f_1(x)) \times \frac{df_1}{du}(x))$$

The underlying logic as to why the formula works for chains of length 2, $\frac{df_2}{du}(x) = \frac{df_2}{du}(f_1(x)) \times \frac{df_1}{du}(x)$, also applies here—as does the lack of intuition from looking at the formula alone!

Diagram

The best way to (literally) see why this formula makes sense is via another box diagram, as shown in Figure 1-10.

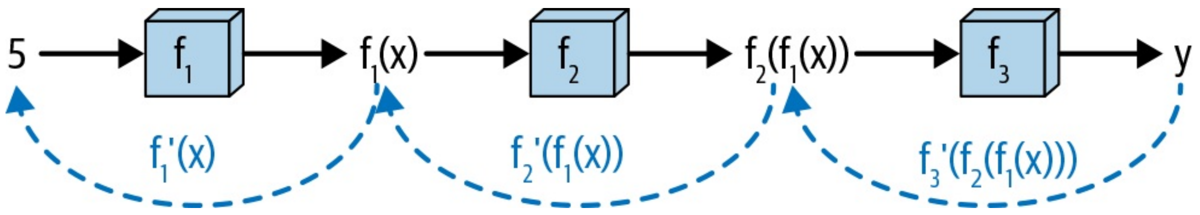


Figure 1-10. The "box model" for computing the derivative of three nested functions

Using similar reasoning to the prior section: if we imagine the input to f_1 f_2 f_3 (call it *a*) being connected to the output (call it *b*) by a string, then changing *a* by a small amount *Δ* will result in a change in $f_1(a)$ of $\frac{df_1}{du}(x))$ times *Δ*, which will result in a change to $f_2(f_1(x))$ (the next step along in the chain) of $\frac{df_2}{du}(f_1(x)) \times \frac{df_1}{du}(x))$ times *Δ*, and so on for the third step, when we get to the final change equal to the full formula for the preceding chain rule times *Δ*.

Spend a bit of time going through this explanation and the earlier diagram—but not too much time, since we'll develop even more intuition for this when we code it up.

Code

How might we translate such a formula into code instructions for computing the derivative, given the constituent functions? Interestingly, already in this simple example we see the beginnings of what will become the forward and backward passes of a neural network:

```python
def chain_deriv_3(chain: Chain,
                  input_range: ndarray) -> ndarray:
    '''
    Uses the chain rule to compute the derivative of three nested functions:
    (f3(f2(f1)))' = f3'(f2(f1(x))) * f2'(f1(x)) * f1'(x)
    '''

    assert len(chain) == 3, \
    "This function requires 'Chain' objects to have length 3"

    f1 = chain[0]
    f2 = chain[1]
    f3 = chain[2]

    # f1(x)
    f1_of_x = f1(input_range)

    # f2(f1(x))
    f2_of_x = f2(f1_of_x)

    # df3du
    df3du = deriv(f3, f2_of_x)

    # df2du
    df2du = deriv(f2, f1_of_x)

    # df1dx
    df1dx = deriv(f1, input_range)

    # Multiplying these quantities together at each point
    return df1dx * df2du * df3du
```

Something interesting took place here—to compute the chain rule for this nested function, we made two "passes" over it:

1. First, we went "forward" through it, computing the quantities `f1_of_x` and `f2_of_x` along the way. We can call this (and think of it as) "the forward pass."
2. Then, we "went backward" through the function, using the quantities that we computed on the forward pass to compute the quantities that make up the derivative.

Finally, we multiplied three of these quantities together to get our derivative.

Now, let's show that this works, using the three simple functions we've defined so far: `sigmoid`, `square`, and `leaky_relu`.

```
PLOT_RANGE = np.range(-3, 3, 0.01)
plot_chain([leaky_relu, sigmoid, square], PLOT_RANGE)
plot_chain_deriv([leaky_relu, sigmoid, square], PLOT_RANGE)
```

Figure 1-11 shows the result.

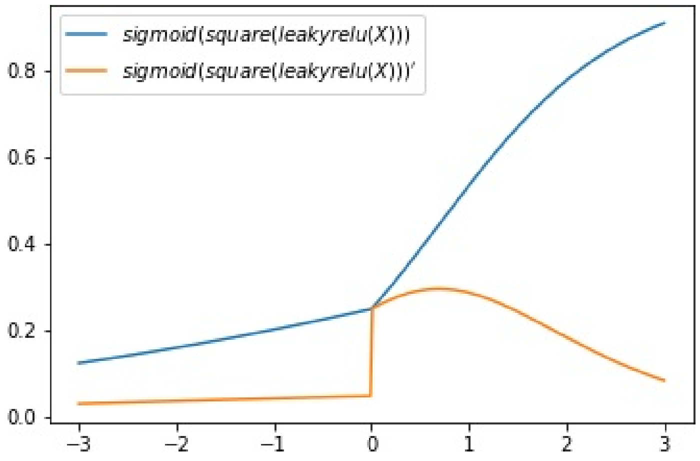


Figure 1-11. The chain rule works, even with triply nested functions

Again, comparing the plots of the derivatives to the slopes of the original functions, we see that the chain rule is indeed computing the derivatives properly.

Let's now apply our understanding to composite functions with multiple inputs, a class of functions that follows the same principles we already established and is ultimately more applicable to deep learning.

Functions with Multiple Inputs

By this point, we have a conceptual understanding of how functions can be strung together to form composite functions. We also have a sense of how to represent these functions as series of boxes that inputs go into and outputs come out of. Finally, we've walked through how to compute the derivatives of these functions so that we understand these derivatives both mathematically and as quantities computed via a step-by-step process with a "forward" and "backward" component.

Oftentimes, the functions we deal with in deep learning don't have just one input. Instead, they have several inputs that at certain steps are added together, multiplied, or otherwise combined. As we'll see, computing the derivatives of the outputs of these functions with respect to their inputs is still no problem: let's consider a very simple scenario with multiple inputs, where two inputs are added together and then fed through another function.

Math

For this example, it is actually useful to start by looking at the math. If our inputs are x and y, then we could think of the function as occurring in two steps. In Step 1, x and y are fed through a function that adds them together. We'll denote that function as α (we'll use Greek letters to refer to function names throughout) and the output of the function as a. Formally, this is simply:

$$a = \alpha(x, y) = x + y$$

Step 2 would be to feed a through some function σ (σ can be any continuous function, such as `sigmoid`, or the `square` function, or even a function whose name doesn't start with s). We'll denote the output of this function as s:

$$s = \sigma(a)$$

We could, equivalently, denote the entire function as f and write:

$$f(x, y) = \sigma(x + y)$$

This is more mathematically concise, but it obscures the fact that this is really two operations happening sequentially. To illustrate that, we need the diagram in the next section.

Diagram

Now that we're at the stage where we're examining functions with multiple inputs, let's pause to define a concept we've been dancing around: the diagrams with circles and arrows connecting them that represent the mathematical "order of operations" can be thought of as *computational graphs*. For example, Figure 1-12 shows a computational graph for the function *f* we just described.

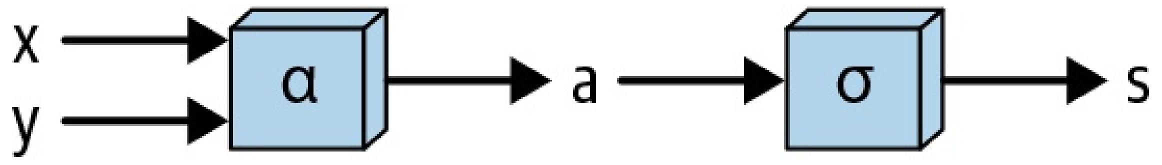


Figure 1-12. Function with multiple inputs

Here we see the two inputs going into *α* and coming out as *a* and then being fed through *σ*.

Code

Coding this up is very straightforward; note, however, that we have to add one extra assertion:

```
def multiple_inputs_add(x: ndarray,
                        y: ndarray,
                        sigma: Array_Function) -> float:
    '''
    Function with multiple inputs and addition, forward pass.
    '''
    assert x.shape == y.shape

    a = x + y
    return sigma(a)
```

Unlike the functions we saw earlier in this chapter, this function does not simply operate "elementwise" on each element of its input `ndarray`s. Whenever we deal with an operation that takes multiple `ndarray`s as inputs, we have to check their shapes to ensure they meet whatever conditions are required by that operation.

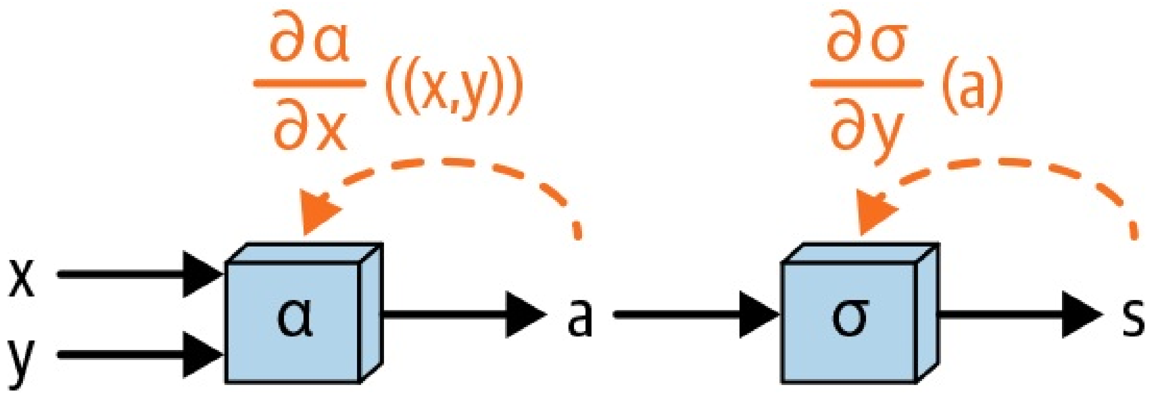
Here, for a simple operation such as addition, all we need to check is that the shapes are identical so that the addition can happen elementwise.

Derivatives of Functions with Multiple Inputs

It shouldn't seem surprising that we can compute the derivative of the output of such a function with respect to both of its inputs.

Diagram

Conceptually, we simply do the same thing we did in the case of functions with one input: compute the derivative of each constituent function "going backward" through the computational graph and then multiply the results together to get the total derivative. This is shown in Figure 1-13.



$\frac{\partial \alpha}{\partial x}((x,y))$
$\frac{\partial \sigma}{\partial y}(a)$
x
y
α
a
σ
s

Figure 1-13. Going backward through the computational graph of a function with multiple inputs

Math

The chain rule applies to these functions in the same way it applied to the functions in the prior sections. Since this is a nested function, with $f(x, y) = \sigma(\alpha(x, y))$, we have:

$$\frac{\partial f}{\partial x} = \frac{\partial \sigma}{\partial u}(\alpha(x, y)) \times \frac{\partial \alpha}{\partial x}((x, y)) = \frac{\partial \sigma}{\partial u}(x + y) \times \frac{\partial \alpha}{\partial x}((x, y))$$

And of course $\frac{\partial f}{\partial y}$ would be identical.

Now note that:

$$\frac{\partial \alpha}{\partial x}((x, y)) = 1$$

since for every unit increase in *x*, *a* increases by one unit, no matter the value of *x* (the same holds for *y*).

Given this, we can code up how we might compute the derivative of such a function.

Code

```
def multiple_inputs_add_backward(x: ndarray,
                                 y: ndarray,
                                 sigma: Array_Function) -> float:
    '''
    Computes the derivative of this simple function with respect to
    both inputs.
    '''
    # Compute "forward pass"
    a = x + y

    # Compute derivatives
    dsda = deriv(sigma, a)

    dadx, dady = 1, 1

    return dsda * dadx, dsda * dady
```

A straightforward exercise for the reader is to modify this for the case where `x` and `y` are multiplied instead of added.

Next, we'll examine a more complicated example that more closely mimics what happens in deep learning: a similar function to the previous example, but with two *vector* inputs.

Functions with Multiple Vector Inputs

In deep learning, we deal with functions whose inputs are *vectors* or *matrices*. Not only can these objects be added, multiplied, and so on, but they can also combined via a dot product or a matrix multiplication. In the rest of this chapter, I'll show how the mathematics of the chain rule and the logic of computing the derivatives of these functions using a forward and backward pass can still apply.

These techniques will end up being central to understanding why deep learning works. In deep learning, our goal will be to fit a model to some data. More precisely, this means that we want to find a mathematical function that maps *observations* from the data—which will be inputs to the function—to some desired *predictions* from the data—which will be the outputs of the function—in as optimal a way as possible. It turns out these observations will be encoded in matrices, typically with row as an observation and each column as a numeric feature for that observation. We'll cover this in more detail in the next chapter; for now, being able to reason about the derivatives of complex functions involving dot products and matrix multiplications will be essential.

Let's start by defining precisely what I mean, mathematically.

Math

A typical way to represent a single data point, or "observation," in a neural network is as a row with n features, where each feature is simply a number x_1, x_2, and so on, up to x_n:

$$X = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix}$$

A canonical example to keep in mind here is predicting housing prices, which we'll build a neural network from scratch to do in the next chapter; in this example, x_1, x_2, and so on are numerical features of a house, such as its square footage or its proximity to schools.

Creating New Features from Existing Features

Perhaps the single most common operation in neural networks is to form a "weighted sum" of these features, where the weighted sum could emphasize certain features and de-emphasize others and thus be thought of as a new feature that itself is just a combination of old features. A concise way to express this mathematically is as a *dot product* of this observation, with some set of "weights" of the same length as the features, w_1, w_2, and so on, up to w_n. Let's explore this concept from the three perspectives we've used thus far in this chapter.

Math

To be mathematically precise, if:

$$W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

then we could define the output of this operation as:

$$N = \nu(X, W) = X \times W = x_1 \times w_1 + x_2 \times w_2 + \ldots + x_n \times w_n$$

Note that this operation is a special case of a *matrix multiplication* that just happens to be a dot product because X has one row and W has only one column.

Next, let's look at a few ways we could depict this with a diagram.

Diagram

A simple way of depicting this operation is shown in Figure 1-14.

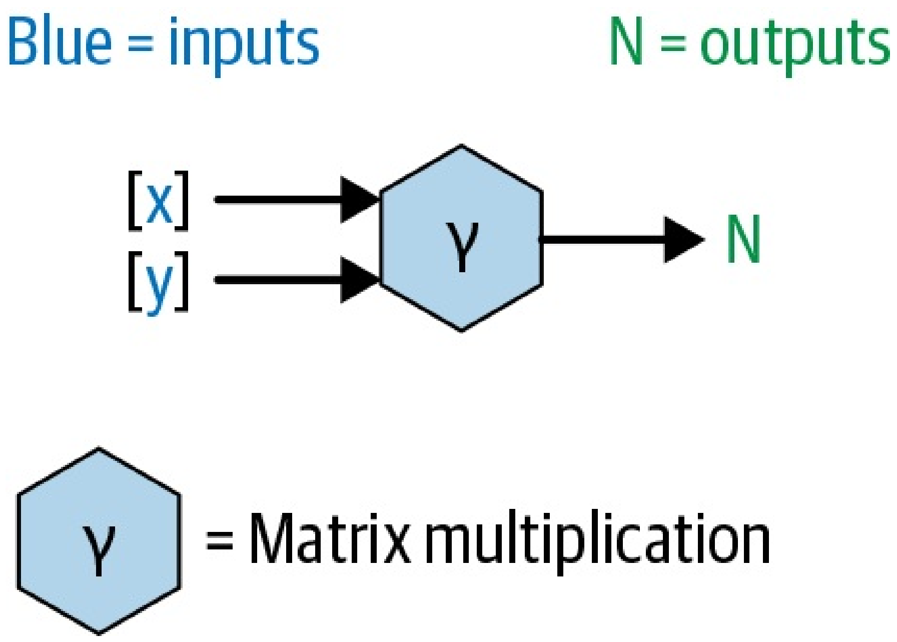


Figure 1-14. Diagram of a vector dot product

This diagram depicts an operation that takes in two inputs, both of which can be `ndarray`s, and produces one output `ndarray`.

But this is really a massive shorthand for many operations that are happening on many inputs. We could instead highlight the individual operations and inputs, as shown in Figures 1-15 and 1-16.

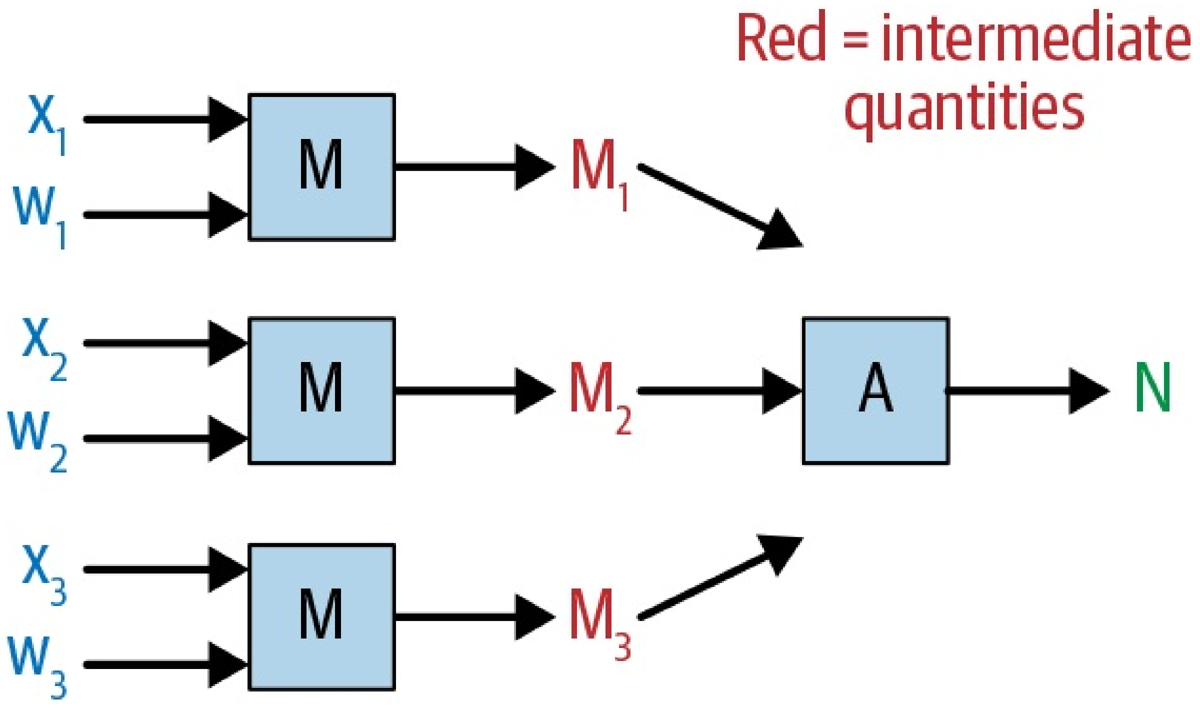


Figure 1-15. Another diagram of a matrix multiplication

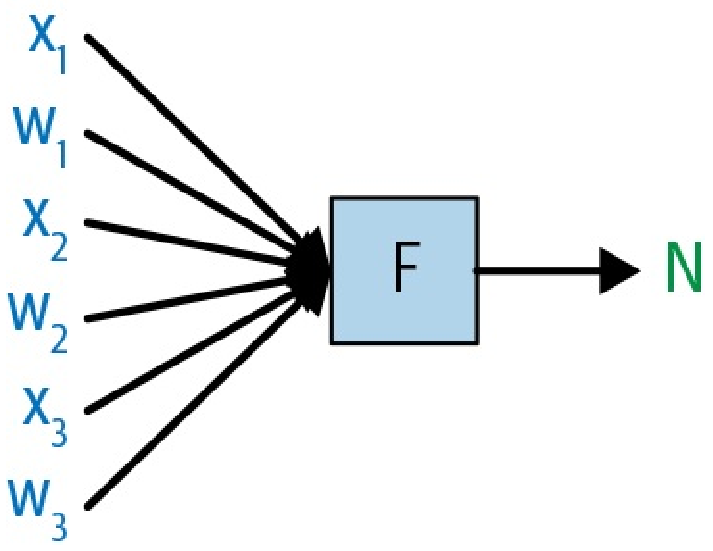


Figure 1-16. A third diagram of a matrix multiplication

The key point is that the dot product (or matrix multiplication) is a concise way to represent many individual operations; in addition, as we'll start to see in the next section, using this operation makes our derivative calculations on the backward pass extremely concise as well.

Code

Finally, in code this operation is simply:

```
def matmul_forward(X: ndarray,
                   W: ndarray) -> ndarray:
    '''
    Computes the forward pass of a matrix multiplication.
    '''

    assert X.shape[1] == W.shape[0], \
    '''
    For matrix multiplication, the number of columns in the first array should
    match the number of rows in the second; instead the number of columns in the
    first array is {0} and the number of rows in the second array is {1}.
    '''.format(X.shape[1], W.shape[0])

    # matrix multiplication
    N = np.dot(X, W)

    return N
```

where we have a new assertion that ensures that the matrix multiplication will work. (This is necessary since this is our first operation that doesn't merely deal with `ndarrays` that are the same size and perform an operation elementwise—our output is now actually a different size than our input.)

Derivatives of Functions with Multiple Vector Inputs

For functions that simply take one input as a number and produce one output, like $f(x) = x^2$ or $f(x) = \text{sigmoid}(x)$, computing the derivative is straightforward: we simply apply rules from calculus. For vector functions, it isn't immediately obvious what the derivative is: if we write a dot product as $\nu(X, W) = N$, as in the prior section, the question naturally arises—what would $\frac{\partial N}{\partial X}$ and $\frac{\partial N}{\partial W}$ be?

Diagram

Conceptually, we just want to do something like in Figure 1-17.

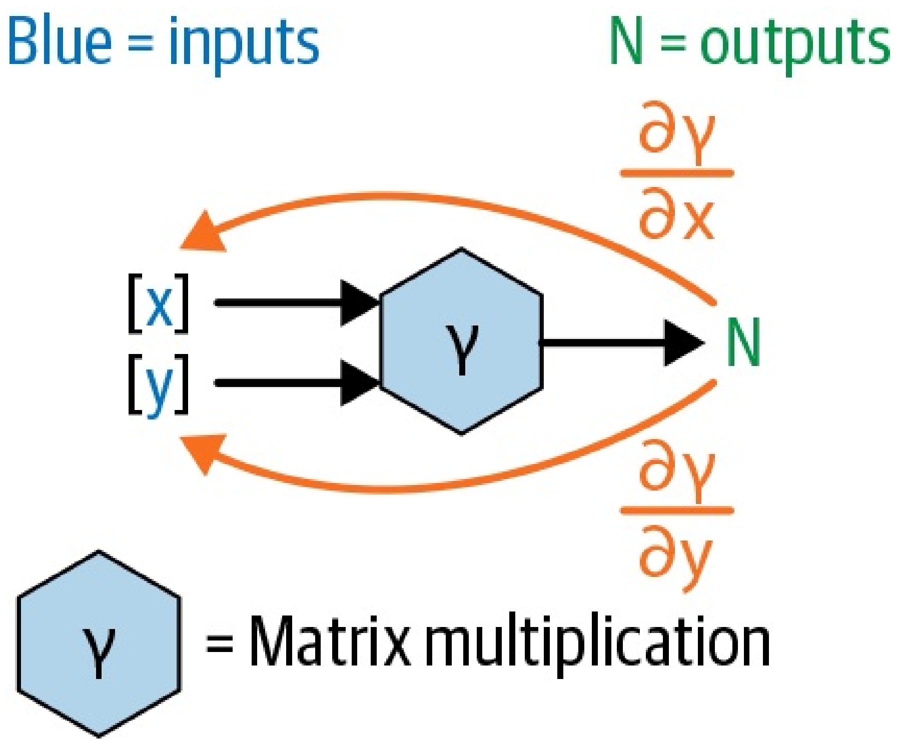

Blue = inputs
N = outputs
$\frac{\partial \nu}{\partial x}$
[x]
[y]
ν
N
$\frac{\partial \nu}{\partial y}$
ν
= Matrix multiplication

Figure 1-17. Backward pass of a matrix multiplication, conceptually