Extracted from:

Programming Machine Learning

From Coding to Deep Learning

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Programming Machine Learning

From Coding to Deep Learning

Paolo Perrotta
edited by Katharine Dvorak

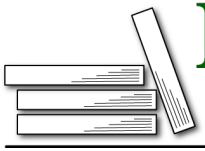
Programming Machine Learning

From Coding to Deep Learning

Paolo Perrotta

The Pragmatic Bookshelf

Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Dave Rankin
Development Editor: Katharine Dvorak
Copy Editor: Jasmine Kwityn
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-660-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: P2.0—March 2021

To my wife Irene,

making my every day.

It's time to make the leap from simple pizza-related examples to the cool stuff: image recognition. We're about to do something magical. Within a few pages, we'll have a program that classifies images.

In this chapter and the next, we'll apply our binary classifier to MNIST, a database of handwritten digits.[1] Just a few years ago, before ML systems could tackle more complex datasets, AI researchers used MNIST as a benchmark for their algorithms. In this chapter, we'll join that esteemed crowd.

"Not MNIST Again!"

Besides being a common benchmark in the industry, MNIST is a bit like the "Hello, World!" of ML tutorials. In fact, if you had previous exposure to ML, you might roll your eyes at the sight of yet another example based on MNIST. Why couldn't I find a more original dataset for this chapter?

Truth is, I chose MNIST *because* it's common, not in spite of that. While most tutorials crack computer vision with the help of high-level ML libraries, we're going to discuss each and every line of code—so we'd better make things easier for ourselves and use a simple, well-documented dataset. Also, a well-known dataset makes it easier for you to look up and compare alternative examples on the Internet.

This will be our first experience with computer vision, so let's start small. We'll begin by recognizing a single MNIST digit, and leave more general character recognition to the next chapter.

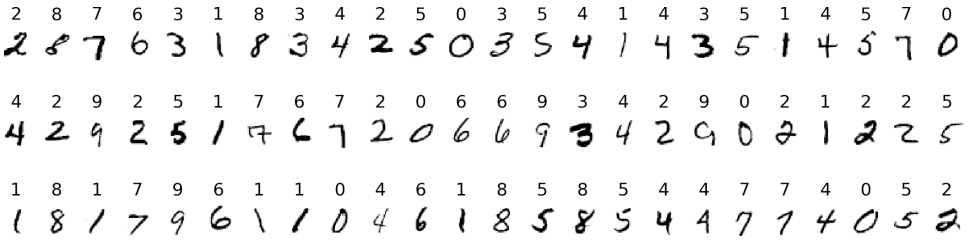
Data Come First

Before we feed data to our ML system, let's get up close and personal with those data. This section tells you all you need to know about MNIST.

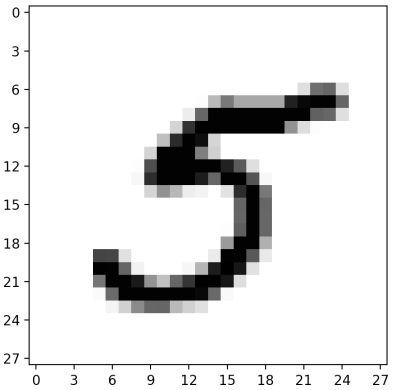
Getting to Know MNIST

MNIST is a collection of labeled images that's been assembled specifically for supervised learning. Its name stands for "Modified NIST," because it's a remix of earlier data from the National Institute of Standards and Technology. MNIST contains images of handwritten digits, labeled with their numerical values. Here are a few random images, capped by their labels:

1. yann.lecun.com/exdb/mnist



Digits are made up of 28 by 28 grayscale pixels, each represented by one byte. In MNIST's grayscale, 0 stands for "perfect background white," and 255 stands for "perfect foreground black." Here's one digit close up:

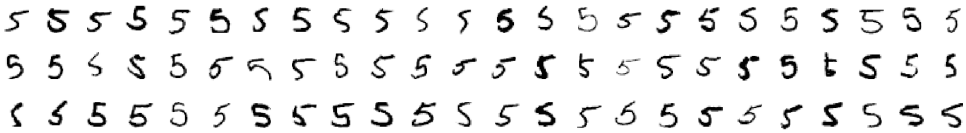


MNIST isn't huge by modern ML standards, but it isn't tiny either. It contains 70,000 examples, neatly partitioned into 7,000 examples for each digit from 0 to 9. Digits have been collected from the wild, so they're quite diverse, as this random assortment of 5s on page 9 proves.

Chalk it up to my age, but I have trouble reading some of these 5s myself. Some look like barely readeable squiggles. If we can write a computer program that recognizes these digits, then we definitely win bragging rights.

Datasets such as MNIST are a godsend to the machine learning community. As cool as ML is, it involves a lot of grindwork to collect, label, clean, and preprocess data. The maintainers of MNIST did that work for us. They collected the images, labeled them, scaled them to the same size, rotated them, centered them, and converted them to the same scale of grays. You can take these digits and feed them straight to a learning program.

MNIST stores images and labels in separate files. There are plenty of libraries that read these data, but the format is simple enough that it makes sense to code our own reader. We'll do that in a few pages.



By the way, you don't have to download the MNIST files: you'll find them among the book's source code, in the `data` directory. Take a peek, and you'll see that MNIST is made up of four files. Two of them contain 60,000 images and their matching labels, respectively. The other two files contain the remaining 10,000 images and labels that have been reserved for testing.

You might wonder why we don't use the same images and labels for both training and testing. The answer to that question requires a short aside.

Training vs. Testing

Consider how we have tested our learning programs so far. First, we trained them on labeled examples. Then we used those same examples to calculate an array of predictions, and we compared the predictions to the ground truth. The closer the two arrays, the better the forecast. That approach served us well as we learned the basics, but it would fail in a real-life project. Here is why.

To make my point, let me come up with an extreme example. Imagine that you're at the pub, chattering about MNIST with a friend. After a few beers, your friend proposes a bet: she will code a system that learns MNIST images in a single iteration of training, and then comes up with 100% accurate classifications. How ludicrous! That sounds like an easy win, but your cunning friend can beat you out in this bet easily. Can you imagine how?

Your friend writes a `train()` function that does nothing, except for storing training examples in a dictionary—a data structure that matches keys to values. (Depending on your coding background, you might call it a "map," a "hashtable," or some other similar name.) In this dictionary, images are keys, and labels are values. Later on, when you ask it to classify an image, the program just looks up that image in the dictionary and returns the matching label. Hey presto, perfectly accurate forecasts in one iteration of training—and without even bothering to implement a machine learning algorithm!

As you pay that beer, you'd be right to grumble that your friend is cheating. Her system didn't really *learn*—it just *memorized* the images and their labels. Confronted with an image that it hasn't seen before, such a system would respond with an awkward silence. Unlike a proper learning system, it wouldn't be able to generalize its knowledge to new data.

Here's a twist: even if nobody is looking to cheat, many ML systems have a built-in tendency to memorize training data. This is a phenomenon known as *overfitting*. You can see that a system is overfitting when it performs better on familiar data than it performs on new data. Such a system could be very accurate at classifying images that it's already seen during training, and then disappoint you when confronted with unfamiliar images.

We'll talk a lot about overfitting in the rest of this book, and we'll even explore a few techniques to reduce its impact. For now, a simple recommendation: *never test a system with the same data that you used to train it.* Otherwise, you might get an unfairly optimistic result, because of overfitting. Before you train the system, set aside a few of your examples for testing, and don't touch them until the training is done.

Now we know how MNIST is organized, and why it's split into separate training and testing set. That's all the knowledge we need. Let's write code that loads MNIST data, and massages those data into a suitable format for our learning program.

Our Own MNIST Library

Let's recap where we are and where we want to go. In the previous chapters, we built a binary classifier. Now we want to apply that program to MNIST.

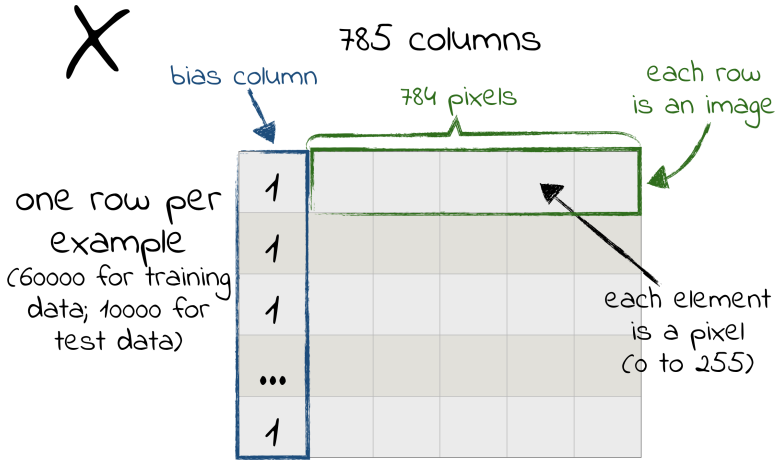
As a first step, we need to reshape MNIST's images and labels into an input for our program. Let's see how to do that.

Preparing the Input Matrices

Our binary classifier program expects its input formatted as two matrices: a set of examples `X`, and a set of labels `Y`. Let's start with the matrix of examples `X`.

`X` is supposed to have one line per example and one column per input variable, plus a *bias column* full of 1s. (Remember the bias column? We talked about it in Bye Bye, Bias, on page ?.)

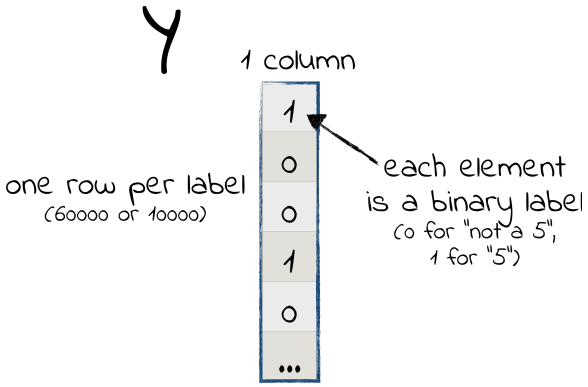
To fit MNIST's images to this format, we can reshape each image to a single line of pixels, so that each pixel becomes an input variable. MNIST images are 28 by 28 pixels, so squashing them results in lines of 784 elements. Throw in the bias column, and that makes 785. So that's what `X` should look like: a matrix of 60,000 lines (the number of examples) and 785 columns (one per pixel, plus the bias). Check out the following graph:



We just graduated from toy examples with three or four input variables to tens of thousands of examples and hundreds of input variables!

Now let's look at `Y`, the matrix of labels. At first glance, it looks simpler than the matrix of images: it still has one line per example, but only one column, that contains the label. However, we have an additional difficulty here: so far we only built binary classifiers, which expect a label that's either 0 or 1. By contrast, MNIST labels range from 0 to 9. How can we fit ten different values into either 0 or 1?

For now, we can work around that problem by narrowing our scope: let's start by recognizing one specific digit—say, the digit 5. This is a problem of binary classification, because a digit can belong to two classes: "not a 5" and "5." This means that we should convert all MNIST labels to 0s, except for 5s, which we should convert to 1s. So that's what our `Y` matrix will look like:



Leap of Faith

In Preparing the Input Matrices, on page 10, we squash each MNIST image into a row of the X matrix. You might have been scratching your head at this idea. Aren't we destroying the images by flattening them? What's the point of having handwritten digits if we grind them into meaningless rows of pixels?

Indeed, we're performing a leap of faith here: we're trusting in the power of statistics. Even though the geometry of those digits is lost, we're betting that the distribution of their pixels is enough information to identify them. For example, the central pixels in a 7 are likely to be darker than the central pixels in a 0. The brightness of a single pixel might be a very weak hint—but by adding up enough of those weak hints, we hope to get a clue of the digit we're looking at.

This method of looking at individual pixels would likely fail on more complicated data than MNIST. For example, imagine writing a classifier that recognizes the species of an elephant from its picture. When they haven't been wallowing in mud, most elephants look gray—so the classifier would probably need more than the pixels' colors to do its job. Toward the end of this book (in Chapter 19, Beyond Vanilla Networks, on page ?), we'll describe a more powerful image recognition algorithm—one that focuses on shapes composed of multiple pixels.

Now we know how to build X and Y. Let's turn this plan into code.

Cooking Data

The Internet overflows with libraries and code snippets that read MNIST data. But we're developers, so hey, let's write one more! In this section I'll show you code for a tiny library that loads those images and labels, and reshapes them into the X and Y that we've just described.

Loading Images

Here is code that loads MNIST images into two matrices—one for the training examples, and one for the test examples:

06_real/mnist.py

```
import numpy as np
import gzip
import struct

def load_images(filename):
    # Open and unzip the file of images:
    with gzip.open(filename, 'rb') as f:
        # Read the header information into a bunch of variables
        _ignored, n_images, columns, rows = struct.unpack('>IIII', f.read(16))
        # Read all the pixels into a NumPy array of bytes:
        all_pixels = np.frombuffer(f.read(), dtype=np.uint8)
```

```
        # Reshape the pixels into a matrix where each line is an image:
        return all_pixels.reshape(n_images, columns * rows)

def prepend_bias(X):
    # Insert a column of 1s in the position 0 of X.
    # ("axis=1" stands for: "insert a column, not a row")
    return np.insert(X, 0, 1, axis=1)

# 60000 images, each 785 elements (1 bias + 28 * 28 pixels)
X_train = prepend_bias(load_images("../data/mnist/train-images-idx3-ubyte.gz"))

# 10000 images, each 785 elements, with the same structure as X_train
X_test = prepend_bias(load_images("../data/mnist/t10k-images-idx3-ubyte.gz"))
```

Let's go through this code quickly. load_images() unzips and decodes images from MNIST's binary files. This function is specific to MNIST's binary format, so you don't really need to understand its details. If you're curious to learn them, know that struct.unpack() reads data from a binary file according to a pattern string. In this case, the pattern is '>IIII', which means "four unsigned integers, encoded with the most significant byte first." The code's comments should help you understand the rest of this function.

load_images() returns a matrix that's either (60000, 784), in the case of the training images, or (10000, 784), in the case of the test images. Those matrices can then be passed to the second function, prepend_bias(), to give them an extra column full of 1s for the bias.

Finally, the last few lines in the code store the training and test images into two constants. The idea is that the client of this library doesn't need to call load_images() and prepend_bias(). Instead, it can import the library (with import mnist) and then refer to these constants (with mnist.X_train and mnist.X_test).

And that's it about the images. Now, the labels.

Loading Labels

This code loads and prepares MNIST's labels:

06_real/mnist.py

```
def load_labels(filename):
    # Open and unzip the file of images:
    with gzip.open(filename, 'rb') as f:
        # Skip the header bytes:
        f.read(8)
        # Read all the labels into a list:
        all_labels = f.read()
        # Reshape the list of labels into a one-column matrix:
        return np.frombuffer(all_labels, dtype=np.uint8).reshape(-1, 1)
```

```
def encode_fives(Y):
    # Convert all 5s to 1, and everything else to 0
    return (Y == 5).astype(int)

# 60K labels, each with value 1 if the digit is a five, and 0 otherwise
Y_train = encode_fives(load_labels("../data/mnist/train-labels-idx1-ubyte.gz"))

# 10000 labels, with the same encoding as Y_train
Y_test = encode_fives(load_labels("../data/mnist/t10k-labels-idx1-ubyte.gz"))
```

load_labels() loads MNIST labels into a NumPy array, and then molds that array into a one-column matrix. Once again, you don't have to understand this code, as you're not likely to load MNIST labels that often—but read the comments if you're curious. (A reminder: reshape(-1, 1) means: "Arrange these data into a matrix with one column, and however many rows you need.") The function returns a matrix with shape (60000, 1) or (10000, 1), depending on whether we're loading the training labels or the test labels.

The matrix returned by load_labels() contains labels from 0 to 9. We can pass that matrix to encode_fives() to turn those labels into binary values. We're not going to live with encode_fives() for long, so I didn't bother parameterizing it. I just hard-coded it to encode the digit 5.

The one line in encode_fives() is typical NumPy code—very terse, to the point of being a bit obscure. To clarify it, (Y == 5) means: "create an array that contains True where Y contains a 5, and False where it doesn't." That array is then converted to an array of integers, so that all True values becomes 1, and False values becomes 0. The end result is a new matrix with the same shape as Y, that contains 1 where Y contains a 5, and 0 elsewhere.

After those functions, the final lines in the code define two constants. We can use them to access the training labels and the test labels, respectively.

With that, our MNIST library is complete. Let's save it as a file (mnist.py), and use it to feed our ML program.