Figure 6-17. Some images of handwritten digits from the MNIST dataset. The learning challenge is to predict the digit from the image.

MNIST was a very important dataset for the development of machine learning methods for computer vision. The dataset is challenging enough that obvious, non-learning methods don't tend to do well. At the same time, MNIST is small enough that experimenting with new architectures doesn't require very large amounts of computing power.

However, the MNIST dataset has mostly become obsolete. The best models achieve near one hundred percent test accuracy. Note that this fact doesn't mean that the problem of handwritten digit recognition is solved! Rather, it is likely that human scientists have overfit architectures to the MNIST dataset and capitalized on its quirks to achieve very high predictive accuracies. As a result, it's no longer good practice to use MNIST to design new deep architectures. That said, MNIST is still a superb dataset for pedagogical purposes.

Loading MNIST

The MNIST codebase is located online on Yann LeCun's website. The download script pulls down the raw file from the website. Notice how the script caches the download so repeated calls to `download()` won't waste effort.

As a more general note, it's quite common to store ML datasets in the cloud and have user code retrieve it before processing for input into a learning algorithm. The Tox21 dataset we accessed via the DeepChem library in Chapter 4 followed this same design pattern. In general, if you would like to host a large dataset for analysis, hosting on the cloud and downloading to a local machine for processing as necessary seems good practice. (This breaks down for very large datasets however, where network transfer times become exorbitantly expensive.) See Example 6-1.

Example 6-1. This function downloads the MNIST dataset

```
def download(filename):
  """Download the data from Yann's website, unless it's already here."""
  if not os.path.exists(WORK_DIRECTORY):
    os.makedirs(WORK_DIRECTORY)
  filepath = os.path.join(WORK_DIRECTORY, filename)
  if not os.path.exists(filepath):
    filepath, _ = urllib.request.urlretrieve(SOURCE_URL + filename, filepath)
    size = os.stat(filepath).st_size
    print('Successfully downloaded', filename, size, 'bytes.')
  return filepath
```

This download checks for the existence of `WORK_DIRECTORY`. If this directory exists, it assumes that the MNIST dataset has already been downloaded. Else, the script uses the `urllib` Python library to perform the download and prints the number of bytes downloaded.

The MNIST dataset is stored as a raw string of bytes encoding pixel values. In order to easily process this data, we need to convert it into a NumPy array. The function `np.frombuffer` provides a convenience that allows the conversion of a raw byte buffer into a numerical array (Example 6-2). As we have noted elsewhere in this book, deep networks can be destabilized by input data that occupies wide ranges. For stable gradient descent, it is often necessary to constrain inputs to span a bounded range. The original MNIST dataset contains pixel values ranging from 0 to 255. For stability, this range needs to be shifted to have mean zero and unit range (from –0.5 to +0.5).

Example 6-2. Extracting images from a downloaded dataset into NumPy arrays

```
def extract_data(filename, num_images):
  """Extract the images into a 4D tensor [image index, y, x, channels].

  Values are rescaled from [0, 255] down to [-0.5, 0.5].
  """
  print('Extracting', filename)
  with gzip.open(filename) as bytestream:
    bytestream.read(16)
    buf = bytestream.read(IMAGE_SIZE * IMAGE_SIZE * num_images * NUM_CHANNELS)
    data = numpy.frombuffer(buf, dtype=numpy.uint8).astype(numpy.float32)
```

```
    data = (data - (PIXEL_DEPTH / 2.0)) / PIXEL_DEPTH
    data = data.reshape(num_images, IMAGE_SIZE, IMAGE_SIZE, NUM_CHANNELS)
    return data
```

The labels are stored in a simple file as a string of bytes. There is a header consisting of 8 bytes, with the remainder of the data containing labels (Example 6-3).

Example 6-3. This function extracts labels from the downloaded dataset into an array of labels

```
def extract_labels(filename, num_images):
  """Extract the labels into a vector of int64 label IDs."""
  print('Extracting', filename)
  with gzip.open(filename) as bytestream:
    bytestream.read(8)
    buf = bytestream.read(1 * num_images)
    labels = numpy.frombuffer(buf, dtype=numpy.uint8).astype(numpy.int64)
  return labels
```

Given the functions defined in the previous examples, it is now feasible to download and process the MNIST training and test dataset (Example 6-4).

Example 6-4. Using the functions defined in the previous examples, this code snippet downloads and processes the MNIST train and test datasets

```
# Get the data.
train_data_filename = download('train-images-idx3-ubyte.gz')
train_labels_filename = download('train-labels-idx1-ubyte.gz')
test_data_filename = download('t10k-images-idx3-ubyte.gz')
test_labels_filename = download('t10k-labels-idx1-ubyte.gz')

# Extract it into NumPy arrays.
train_data = extract_data(train_data_filename, 60000)
train_labels = extract_labels(train_labels_filename, 60000)
test_data = extract_data(test_data_filename, 10000)
test_labels = extract_labels(test_labels_filename, 10000)
```

The MNIST dataset doesn't explicitly define a validation dataset for hyperparameter tuning. Consequently, we manually designate the final 5,000 datapoints of the training dataset as validation data (Example 6-5).

Example 6-5. Extract the final 5,000 datasets of the training data for hyperparameter validation

```
VALIDATION_SIZE = 5000  # Size of the validation set.

# Generate a validation set.
validation_data = train_data[:VALIDATION_SIZE, ...]
```

```
validation_labels = train_labels[:VALIDATION_SIZE]
train_data = train_data[VALIDATION_SIZE:, ...]
train_labels = train_labels[VALIDATION_SIZE:]
```

Choosing the Correct Validation Set

In Example 6-5, we use the final fragment of training data as a validation set to gauge the progress of our learning methods. In this case, this method is relatively harmless. The distribution of data in the test set is well represented by the distribution of data in the validation set.

However, in other situations, this type of simple validation set selection can be disastrous. In molecular machine learning (the use of machine learning to predict properties of molecules), it is almost always the case that the test distribution is dramatically different from the training distribution. Scientists are most interested in *prospective* prediction. That is, scientists would like to predict the properties of molecules that have never been tested for the property at hand. In this case, using the last fragment of training data for validation, or even a random subsample of the training data, will lead to misleadingly high accuracies. It's quite common for a molecular machine learning model to have 90% accuracy on validation and, say, 60% on test.

To correct for this error, it becomes necessary to design validation set selection methods that take pains to make the validation dissimilar from the training set. A variety of such algorithms exist for molecular machine learning, most of which use various mathematical estimates of graph dissimilarity (treating a molecule as a mathematical graph with atoms as nodes and chemical bonds as edges).

This issue crops up in many other areas of machine learning as well. In medical machine learning or in financial machine learning, relying on historical data to make forecasts can be disastrous. For each application, it's important to critically reason about whether performance on the selected validation set is actually a good proxy for true performance.

TensorFlow Convolutional Primitives

We start by introducing the TensorFlow primitives that are used to construct our convolutional networks (Example 6-6).