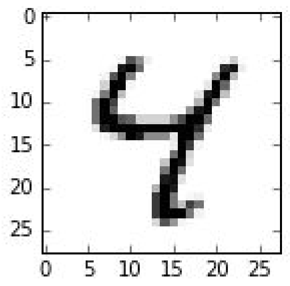
Recognising human handwriting is an ideal challenge for testing artificial intelligence, because the problem is sufficiently hard and fuzzy. It's not clear and defined like multiplying lots of lots of numbers.

Getting computers to correctly classify what an image contains, sometimes called the **image recognition** problem, has withstood decades of attack. Only recently has good progress been made, and methods like neural networks have been a crucial part of these leaps forward.

To give you a sense of how hard the problem of image recognition is, we humans will sometimes disagree on what an image contains. We'll easily disagree what a handwritten character actually is, particularly if the character was written in a rush or without care. Have a look at the following handwritten number. Is it a 4 or a 9?



There is a collection of images of handwritten numbers used by artificial intelligence researchers as a popular set to test their latest ideas and algorithms. The fact that the collection is well known and popular means that it is easy to check how well our latest crazy idea for image recognition works compared to others. That is, different ideas and algorithms are tested against the same data set.

That data set is called the MNIST database of handwritten digits, and is available from the respected neural network researcher Yann LeCun's website http://yann.lecun.com/exdb/mnist/. That page also lists how well old and new ideas have performed in learning and correctly classifying these handwritten characters. We'll come back to that list several times to see how well our own ideas perform against professionals!

The format of the MNIST database isn't the easiest to work with, so others have helpfully created data files of a simpler format, such as this one http://pjreddie.com/projects/mnist-in-csv/. These files are called CSV files, which means each value is plain text separated by commas (comma separated values). You can easily view them in any text editor, and most spreadsheet or data analysis software will work with CSV files. They are pretty much a universal standard. This website provides two CSV files:

* A **training** set http://www.pjreddie.com/media/files/mnist_train.csv

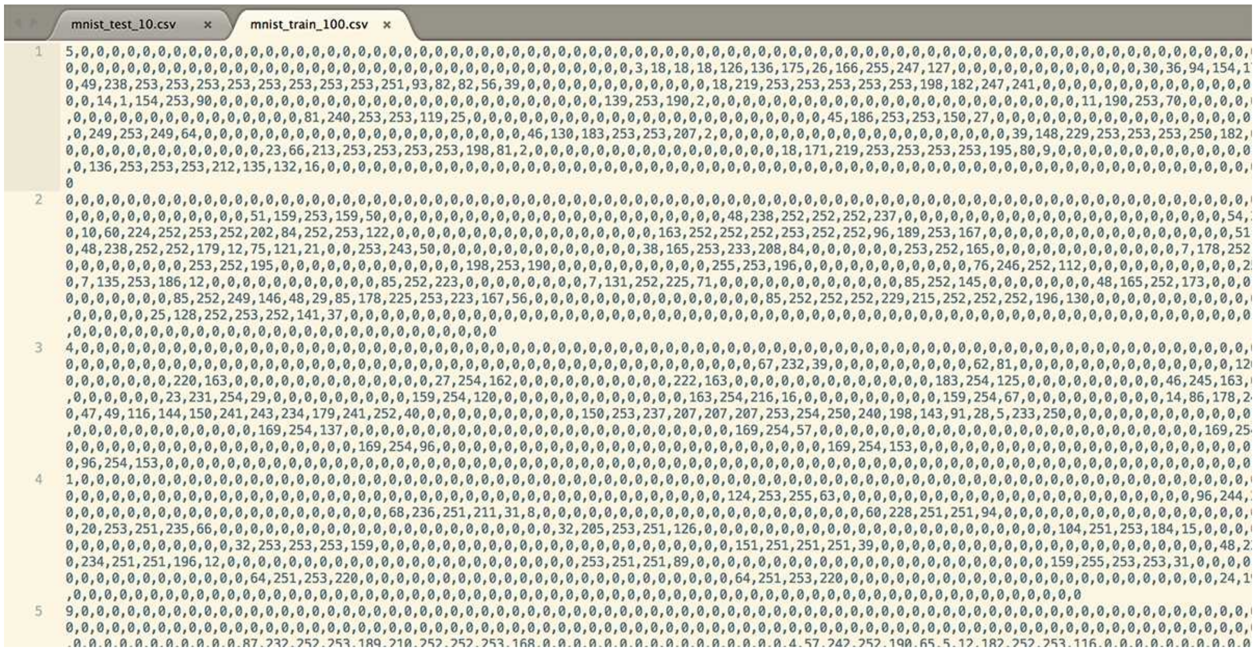
* A **test** set http://www.pjreddie.com/media/files/mnist_test.csv

As the names suggest, the **training set** is the set of **60,000** labelled examples used to train the neural network. **Labelled** means the inputs come with the desired output, that is, what the answer should be.

The smaller **test set** of **10,000** is used to see how well our idea or algorithm works. This too contains the correct labels so we can check to see if our own neural network got the answer right or not.

The idea of having separate training and test data sets is to make sure we test against data we haven't seen before. Otherwise we could cheat and simply memorise the training data to get a perfect, albeit deceptive, score. This idea of separating training from test data is common across machine learning.

Let's take a peek at these files. The following shows a section of the MNIST test set loaded into a text editor.



```
5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3,18,18,18,126,136,175,26,166,255,247,127,0,0,0,0,0,0,0,0,0,0,0,0,30,36,94,154,1
0,49,238,253,253,253,253,253,253,253,253,251,93,82,82,56,39,0,0,0,0,0,0,0,0,0,0,0,0,18,219,253,253,253,253,253,198,182,247,241,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,14,1,154,253,90,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,139,253,190,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,11,190,253,70,0,0,0,0,0,
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,81,240,253,253,119,25,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,45,186,253,253,150,27,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
,0,249,253,249,64,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,46,130,183,253,253,207,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,39,148,229,253,253,253,250,182,
0,0,0,0,0,0,0,0,0,0,0,0,0,23,66,213,253,253,253,253,198,81,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,18,171,219,253,253,253,253,195,80,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
,0,136,253,253,253,212,135,132,16,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,51,159,253,159,50,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,48,238,252,252,252,237,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,54,
0,10,60,224,252,253,252,202,84,252,253,122,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,163,252,252,252,253,252,252,96,189,253,167,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,51,
0,48,238,252,252,179,12,75,121,21,0,0,253,243,50,0,0,0,0,0,0,0,0,0,0,0,0,0,0,38,165,253,233,208,84,0,0,0,0,0,0,253,252,165,0,0,0,0,0,0,0,0,0,0,0,7,178,252
0,0,0,0,0,0,0,0,253,252,195,0,0,0,0,0,0,0,0,0,0,0,0,0,0,198,253,190,0,0,0,0,0,0,0,0,255,253,196,0,0,0,0,0,0,0,0,0,0,76,246,252,112,0,0,0,0,0,0,0,0,0,2
0,7,135,253,186,12,0,0,0,0,0,0,0,0,0,0,85,252,223,0,0,0,0,0,0,0,7,131,252,225,71,0,0,0,0,0,0,0,0,0,0,0,85,252,145,0,0,0,0,0,0,48,165,252,173,0,0,0
0,0,0,0,0,0,0,85,252,249,146,48,29,85,178,225,253,223,167,56,0,0,0,0,0,0,0,0,0,0,0,0,0,0,85,252,252,252,229,215,252,252,252,196,130,0,0,0,0,0,0,0,0,0,0,0,
,0,0,0,0,0,25,128,252,253,252,141,37,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,67,232,39,0,0,0,0,0,0,0,0,0,62,81,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,12
0,0,0,0,0,0,0,220,163,0,0,0,0,0,0,0,0,0,0,0,27,254,162,0,0,0,0,0,0,0,0,0,0,0,0,0,222,163,0,0,0,0,0,0,0,0,0,0,0,0,183,254,125,0,0,0,0,0,0,0,0,0,46,245,163,
,0,0,0,0,0,0,23,231,254,29,0,0,0,0,0,0,0,0,159,254,120,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,163,254,216,16,0,0,0,0,0,0,0,0,159,254,67,0,0,0,0,0,0,0,0,14,86,178,2
0,47,49,116,144,150,241,243,234,179,241,252,40,0,0,0,0,0,0,0,0,0,150,253,237,207,207,207,253,254,250,240,198,143,91,28,5,233,250,0,0,0,0,0,0,0,0,0,0,0,0,0,
,0,0,0,0,0,0,0,0,0,0,0,0,169,254,137,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,169,254,57,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,169,25
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,169,254,96,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,169,254,153,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,96,254,153,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,124,253,255,63,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,96,244,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,68,236,251,211,31,8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,60,228,251,251,94,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,20,253,251,235,66,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,32,205,253,251,126,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,104,251,253,184,15,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,32,253,253,253,159,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,151,251,251,251,39,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,48,2
0,234,251,251,196,12,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,253,251,251,89,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,159,255,253,253,31,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,64,251,253,220,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,24,1
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
,0,0,0,0,0,0,0,0,0,0,0,87,232,252,253,189,210,252,252,253,168,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4,57,242,252,190,65,5,12,182,252,253,116,0,0,0,0,0,0,0,0,0,0,
```

Whoah! That looks like something went wrong! Like one of those movies from the 80s where a computer gets hacked.

Actually all is well. The text editor is showing long lines of text. Those lines consist of numbers, separated by commas. That is easy enough to see. The lines are quite long so they wrap around a few times. Helpfully this text editor shows the real line numbers in the margin, and we can see four whole lines of data, and part of the fifth one.

The content of these records, or lines of text, is easy to understand:

- The first value is the **label**, that is, the actual digit that the handwriting is supposed to represent, such as a "7" or a "9". This is the answer the neural network is trying to learn to get right.

- The subsequent values, all comma separated, are the pixel values of the handwritten digit. The size of the pixel array is 28 by 28, so there are 784 values after the label. Count them if you really want!

So that first record represents the number “5” as shown by the first value, and the rest of the text on that line are the pixel values for someone’s handwritten number 5. The second record represents a handwritten “0”, the third represents “4”, the fourth record is “1” and the fifth represents “9”. You can pick any line from the MNIST data files and the first number will tell you the label for the following image data.

But it is hard to see how that long list of 784 values makes up a picture of someone’s handwritten number 5. We should plot those numbers as an image to confirm that they really are the colour values of handwritten number.

Before we dive in and do that we should download a smaller subset of the MNIST data set. The MNIST data data files are pretty big and working with a smaller subset is helpful because it means we can experiment, trial and develop our code without being slowed down by a large data set slowing our computers down. Once we’ve settled on an algorithm and code we’re happy with, we can use the full data set.

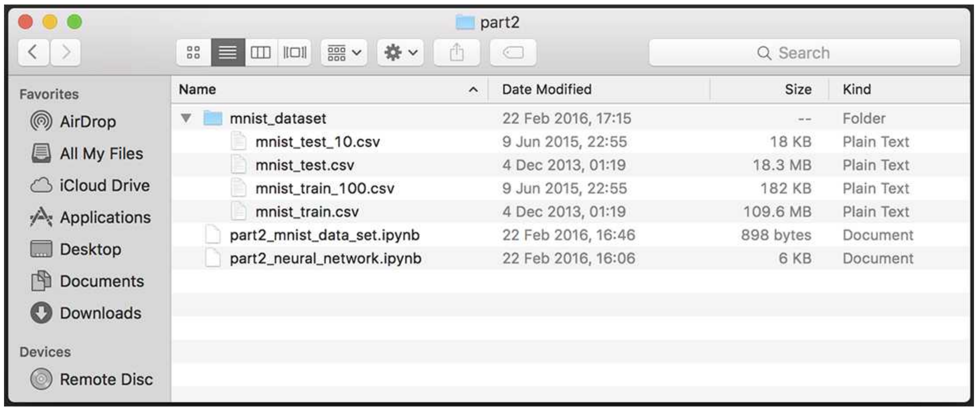
The following are the links to a smaller subsets of the MNIST dataset, also in CSV format:

- 10 records from the MNIST test data set - https://raw.githubusercontent.com/makeyourownneuralnetwork/makeyourownneuralnetwork/master/mnist_dataset/mnist_test_10.csv

- 100 records from the MNIST training data set - https://raw.githubusercontent.com/makeyourownneuralnetwork/makeyourownneuralnetwork/master/mnist_dataset/mnist_train_100.csv

If your browser shows the data instead of downloading it automatically, you can manually save the file using the “File -> Save As …”, or equivalent action in your browser.

Save the data files to a location that works well for you. I keep my data files in a folder called “mnist_dataset” next to my IPython notebooks as shown in the following screenshot. It gets messy if IPython notebooks and data files are scattered all over the place.



Name	Date Modified	Size	Kind
mnist_dataset	22 Feb 2016, 17:15	--	Folder
mnist_test_10.csv	9 Jun 2015, 22:55	18 KB	Plain Text
mnist_test.csv	4 Dec 2013, 01:19	18.3 MB	Plain Text
mnist_train_100.csv	9 Jun 2015, 22:55	182 KB	Plain Text
mnist_train.csv	4 Dec 2013, 01:19	109.6 MB	Plain Text
part2_mnist_data_set.ipynb	22 Feb 2016, 16:46	898 bytes	Document
part2_neural_network.ipynb	22 Feb 2016, 16:06	6 KB	Document

Before we can do anything with the data, like plotting it or training a neural network with it, we need to find a way to get at it from our Python code.

Opening a file and getting its content is really easy in Python. It’s best to just show it and explain it. Have a look at the following code:

```
data_file = open("mnist_dataset/mnist_train_100.csv", 'r')
data_list = data_file.readlines()
data_file.close()
```

There are only three lines of code here. Let’s talk through each one.

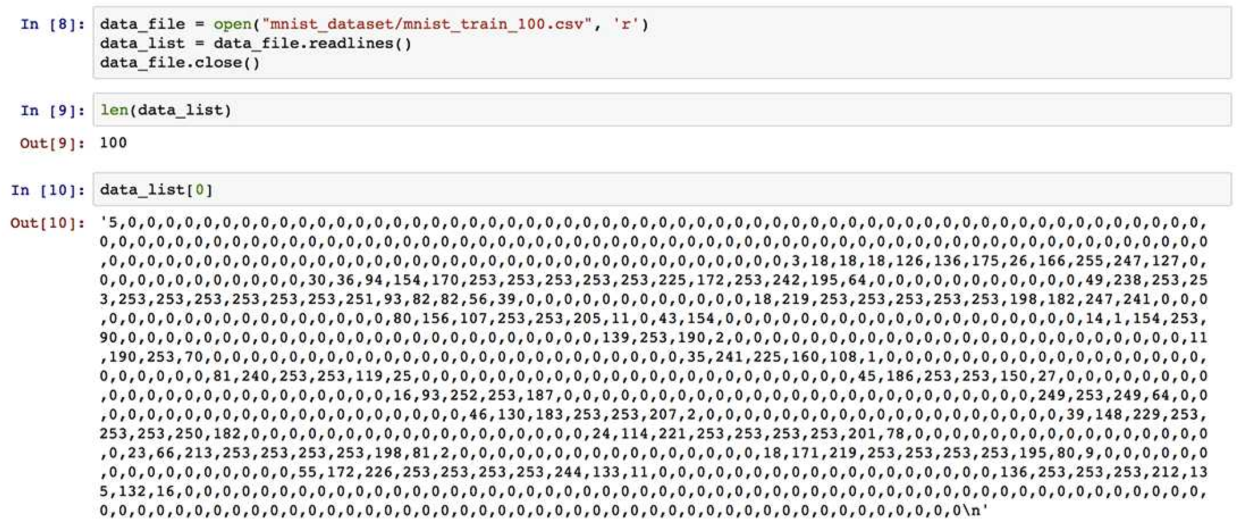
The first line uses a function open() to open a file. You can see first parameter passed to the function is the name of the file. Actually, it is more than just the filename “mnist_train_100.csv”, it is the whole path which includes the directory the file is in. The second parameter is optional, and tells Python how we want to treat the file. The ‘r’ tells Python that we want to open the file for reading only, and not for writing. That way we avoid any accidents changing or deleting the data. If we tried to write to that file and change it, Python would stop us and raise an error. What’s that variable **data_file**? The open() function creates a file handle, a reference, to that file and we’ve assigned it to a variable named **data_file**. Now that we’ve opened the file, any further actions like reading from it, are done through that handle.

The next line is simple. We use the readlines() function associated with the file handle **data_file**, to read all of the lines in the file into the variable data_list. The variable contains a list, where each item of the list is a string representing a line in the file. That’s much more useful because we can jump to specific lines just like we would jump to specific entries in a list. So **data_list**[0] is the first record, and **data_list**[9] is the tenth record, and so on.

By the way, you may hear people tell you not to use readlines() because it reads the entire file into memory. They will tell you to read one line at a time and do whatever work you need with each line, and then move onto the next line. They aren't wrong, it is more efficient to work on a line at a time, and not read the entire file into memory. However our files aren't that massive, and the code is easier if we use readlines(), and for us, simplicity and clarity is important as we learn Python.

The last line closes the file. It is good practice to close and clean up after using resources like files. If we don't, they remain open and can cause problems. What problems? Well, some programs might not want to write to a file that was opened elsewhere in case it led to an inconsistency. It would be like two people trying to write a letter on the same piece of paper! Sometimes your computer may lock a file to prevent this kind of clash. If you don't clean up after you use files, you can have a build up of locked files. At the very least, closing a file, allows your computer to free up memory it used to hold parts of that file.

Create a new empty notebook and try this code, and see what happens when you print out elements of that list a. The following shows this working.



```
In [8]:  data_file = open("mnist_dataset/mnist_train_100.csv", 'r')
         data_list = data_file.readlines()
         data_file.close()

In [9]:  len(data_list)

Out[9]:  100

In [10]: data_list[0]

Out[10]: '5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
         0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
         ,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3,18,18,18,126,136,175,26,166,255,247,127,0,
         0,0,0,0,0,0,0,0,0,0,0,30,36,94,154,170,253,253,253,253,253,225,172,253,242,195,64,0,0,0,0,0,0,0,0,0,0,0,49,238,253,25
         3,253,253,253,253,253,253,251,93,82,82,56,39,0,0,0,0,0,0,0,0,0,0,0,0,18,219,253,253,253,253,253,198,182,247,241,0,0,0
         ,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,80,156,107,253,253,205,11,0,43,154,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,14,1,154,253,
         90,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,139,253,190,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,11
         ,190,253,70,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,35,241,225,160,108,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
         0,0,0,0,0,0,81,240,253,253,119,25,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,45,186,253,253,150,27,0,0,0,0,0,0,0,0
         ,0,0,0,0,0,0,0,0,0,0,0,0,0,0,16,93,252,253,187,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,249,253,249,64,0,0
         ,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,46,130,183,253,253,207,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,39,148,229,253,
         253,253,250,182,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,24,114,221,253,253,253,253,201,78,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
         ,0,23,66,213,253,253,253,253,198,81,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,18,171,219,253,253,253,253,195,80,9,0,0,0,0,0,0
         ,0,0,0,0,0,0,0,0,0,0,55,172,226,253,253,253,253,244,133,11,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,136,253,253,253,212,13
         5,132,16,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
         0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0\n'
```

You can see that the length of the list is 100. The Python len() function tells us how big a list is. You can also see the content of the first record **data_list**[0]. The first number is '5' which is the label, and the rest of the 784 numbers are the colour values for the pixels that make up the image. If you look closely you can tell these colour values seem to range between 0 and 255. You might want to look at other records and see if that is true there too. You'll find that the colour values do indeed fall within the range from 0 to 255.

We did see earlier how we might plot a rectangular array of numbers using the imshow() function. We want to do the same here but we need to convert that list of comma separated numbers into a suitable array. Here are the steps to do that:

- Split that long text string of comma separated values into individual values, using the commas as the place to do the splitting.

- Ignore the first value, which is the label, and take the remaining list of 28 * 28 = 784 values and turn them into an array which has a shape of 28 rows by 28 columns.

- Plot that array!

Again, it is easiest to show the fairly simple Python code that does this, and talk through the code to explain in more detail what is happening.

First we mustn't forget to import the Python extension libraries which will help us with arrays and plotting:

```
import numpy
import matplotlib.pyplot
%matplotlib inline
```

Look at the following 3 lines of code. The variables have been coloured to make it easier to understand which data is being used where.

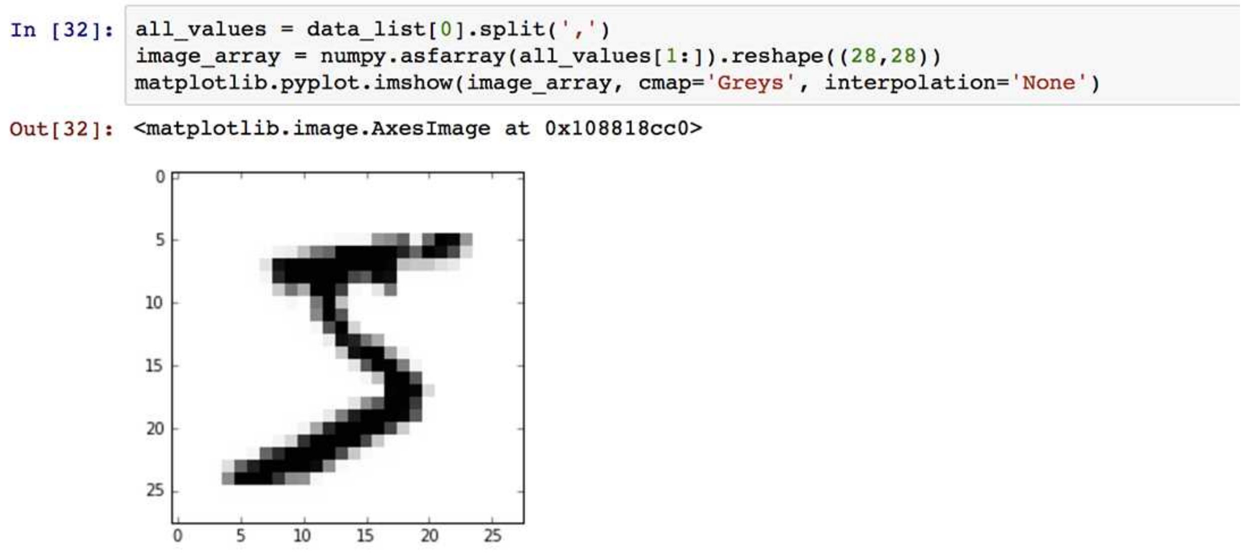
```
all_values = data_list[0].split(',')
image_array = numpy.asfarray(all_values[1:]).reshape((28,28))
matplotlib.pyplot.imshow(image_array, cmap='Greys',
interpolation='None')
```

The first line takes the first records **data_list**[0] that we just printed out, and splits that long string by its commas. You can see the split() function doing this, with a parameter telling it which symbol to split by. In this case that symbol is the comma. The results will be placed into **all_values**. You can print this out to check that is indeed a long Python list of values.

The next line looks more complicated because there are several things happening on the same line. Let's work out from the core. The core has that **all_values** list but this time the square brackets [1:] are used to take all except the first element of this list. This is how we ignore the first label value and only take the rest of the 784 values. The numpy.asfarray() is a numpy function to convert the text strings into real numbers and to create an array of those numbers. Hang on - what do we mean by converting text string into numbers? Well the file was read as text, and each line or record is still text. Splitting each line by the commas still results in bits of text. That text could be the word "apple", "orange123" or "567". The text string "567" is not the same as a number 567. That’s why we need to convert text strings to numbers, even if the text looks like numbers. The last bit .reshape((28,28)) makes sure the list of number is wrapped around every 28 elements to make a square matrix 28 by 28. The resulting 28 by 28 array is called image_array. Phew! That was a fair bit happening in one line.

The third line simply plots the image_array using the imshow() function just like we saw earlier. This time we did select a greyscale colour palette with cmap=’Greys’ to better show the handwritten characters.

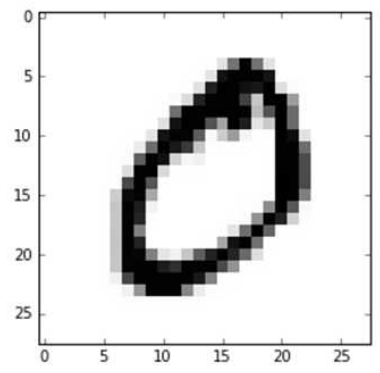
The following shows the results of this code:



```
In [32]: all_values = data_list[0].split(',')
         image_array = numpy.asfarray(all_values[1:]).reshape((28,28))
         matplotlib.pyplot.imshow(image_array, cmap='Greys', interpolation='None')

Out[32]: <matplotlib.image.AxesImage at 0x108818cc0>
```

You can see the plotted image is a 5, which is what the label said it should be. If we instead choose the next record data_list[1] which has a label of 0, we get the following image.

```
In [37]: all_values = data_list[1].split(',')
         image_array = numpy.asfarray(all_values[1:]).reshape((28,28))
         matplotlib.pyplot.imshow(image_array, cmap='Greys', interpolation='None')

Out[37]: <matplotlib.image.AxesImage at 0x108bc3160>
```



You can tell easily the handwritten number is indeed zero.

Preparing the MNIST Training Data

We've worked out how to get data out of the MNIST data files and disentangle it so we can make sense of it, and visualise it too. We want to train our neural network with this data, but we need to think just a little about preparing this data before we throw it at our neural network.

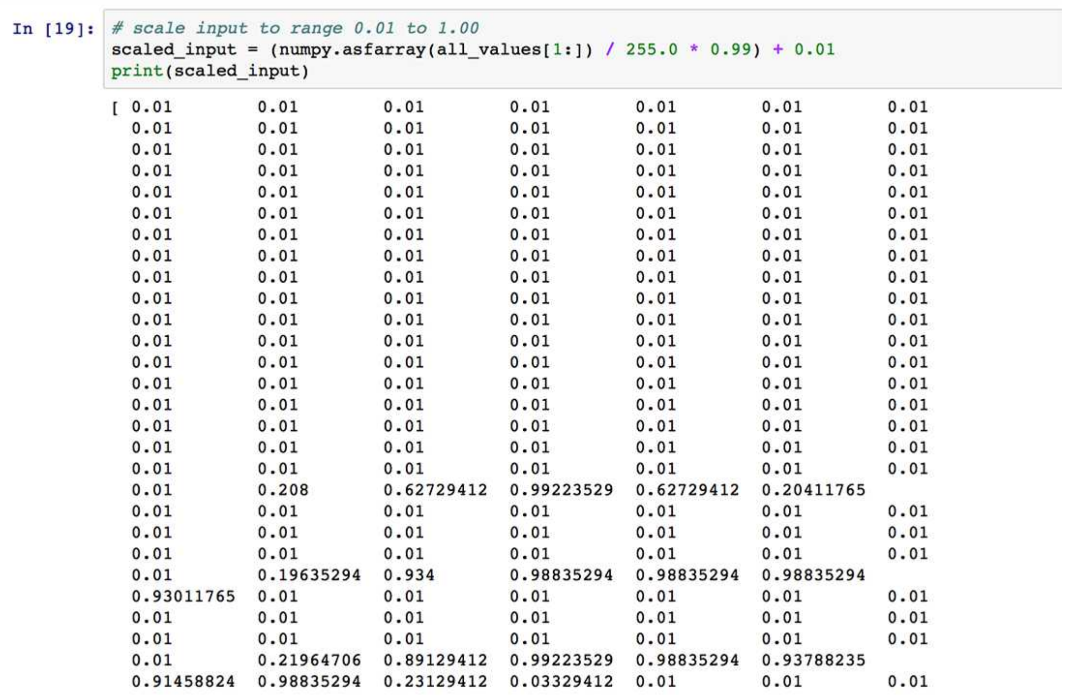
We saw earlier that neural networks work better if the input data, and also the output values, are of the right shape so that they stay within the comfort zone of the network node activation functions.

The first thing we need to do is to rescale the input colour values from the larger range 0 to 255 to the much smaller range 0.01 - 1.0. We've deliberately chosen 0.01 as the lower end of the range to avoid the problems we saw earlier with zero valued inputs because they can artificially kill weight updates. We don't have to choose 0.99 for the upper end of the input because we don't need to avoid 1.0 for the inputs. It's only for the outputs that we should avoid the impossible to reach 1.0.

Dividing the raw inputs which are in the range 0-255 by 255 will bring them into the range 0-1. We then need to multiply by 0.99 to bring them into the range 0.0 - 0.99. We then add 0.01 to shift them up to the desired range 0.01 to 1.00. The following Python code shows this in action:

```
scaled_input = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) +
0.01
print(scaled_input)
```

The output confirms that the values are now in the range 0.01 to 0.99.



```
In [19]: # scale input to range 0.01 to 1.00
         scaled_input = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
         print(scaled_input)
```

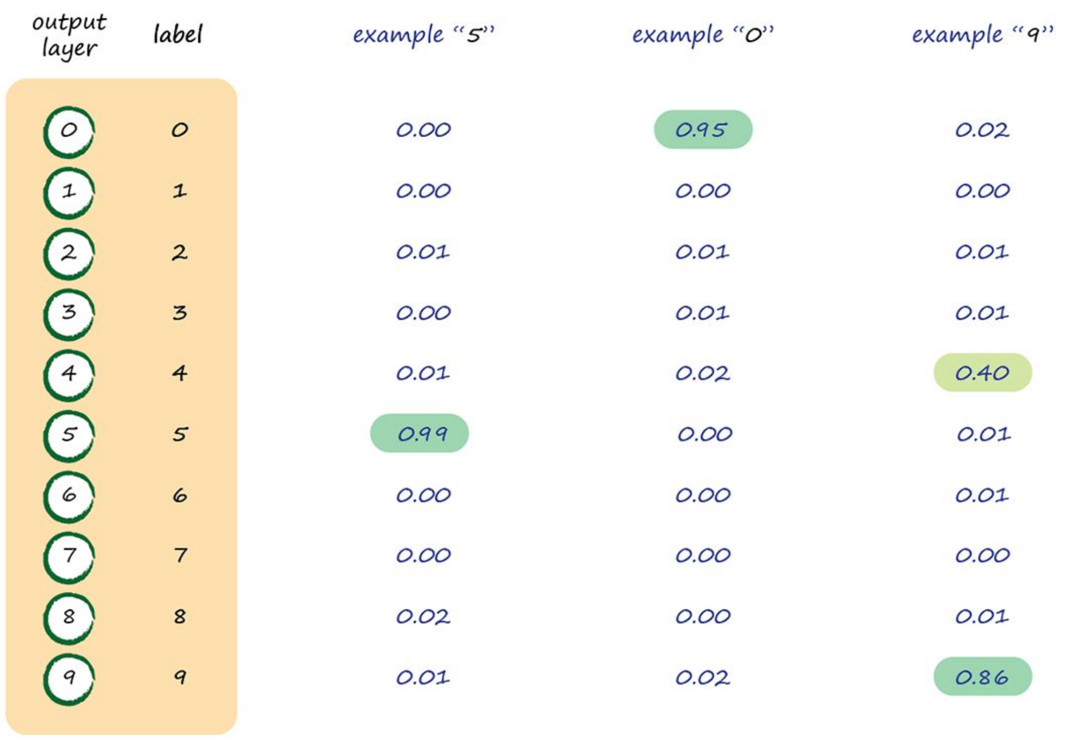
```
[ 0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.208       0.62729412  0.99223529  0.62729412  0.20411765
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.19635294  0.934       0.98835294  0.98835294  0.98835294
  0.93011765  0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.01        0.01        0.01        0.01        0.01        0.01
  0.01        0.21964706  0.89129412  0.99223529  0.98835294  0.93788235
  0.91458824  0.98835294  0.23129412  0.03329412  0.01        0.01        0.01
```

So we've prepared the MNIST data by rescaling and shifting it, ready to throw at our neural network for both training and querying.

We now need to think about the neural network's outputs. We saw earlier that the outputs should match the range of values that the activation function can push out. The logistic function we're using can't push out numbers like -2.0 or 255. The range is between 0.0 and 1.0, and in fact you can't reach 0.0 or 1.0 as the logistic function only approaches these extremes without actually getting there. So it looks like we'll have to scale our target values when training.

But actually, we have a deeper question to ask ourselves. What should the output even be? Should it be an image of the answer? That would mean we have a 28x28 = 784 output nodes.

If we take a step back and think about what we're asking the neural network, we realise we're asking it to classify the image and assign the correct label. That label is one of 10 numbers, from 0 to 9. That means should be able to have an output layer of 10 nodes, one for each of the possible answers, or labels. If the answer was "0" the first output layer node would fire and the rest should be silent. If the answer was "9" the last output layer node would fire and the rest would be silent. The following illustrates this scheme, with some example outputs too.



output layer	label	example "5"	example "0"	example "9"
0	0	0.00	0.95	0.02
1	1	0.00	0.00	0.00
2	2	0.01	0.01	0.01
3	3	0.00	0.01	0.01
4	4	0.01	0.02	0.40
5	5	0.99	0.00	0.01
6	6	0.00	0.00	0.01
7	7	0.00	0.00	0.00
8	8	0.02	0.00	0.01
9	9	0.01	0.02	0.86

The first example is where the neural network thinks it has seen the number “5”. You can see that the largest signal emerging from the output layer is from the node with label 5. Remember this is the sixth node because we’re starting from a label for zero. That’s easy enough. The rest of the output nodes produce a small signal very close to zero. Rounding errors might result in an output of zero, but in fact you’ll remember the activation function doesn’t produce an actual zero.

The next example shows what might happen if the neural network thinks it has seen a handwritten “zero”. Again the largest output by far is from the first output node corresponding to the label “0’.

The last example is more interesting. Here the neural network has produced the largest output signal from the last node, corresponding to the label “9”. However it has a moderately big output from the node for “4”. Normally we would go with the biggest signal, but you can see how the network partly believed the answer could have been “4”. Perhaps the handwriting made it hard to be sure? This sort of uncertainty does happen with neural networks, and rather than see it as a bad thing, we should see it as a useful insight into how another answer was also a contender.

That’s great! Now we need to turn these ideas into target arrays for the neural network training. You can see that if the label for a training example is “5”, we need to create a target array for the output node where all the elements are small except the one corresponding to the label “5”. That could look like the following [0, 0, 0, 0, 0, 1, 0, 0, 0, 0].

In fact we need to rescale those numbers because we've already seen how trying to get the neural network to create outputs of 0 and 1, which are impossible for the activation function, will drive large weights and a saturated network. So we'll use the values 0.01 and 0.99 instead, so the target for the label "5" should be [0.01, 0.01, 0.01, 0.01, 0.01, 0.99, 0.01, 0.01, 0.01, 0.01].

Have a look at the following Python code which constructs the target matrix:

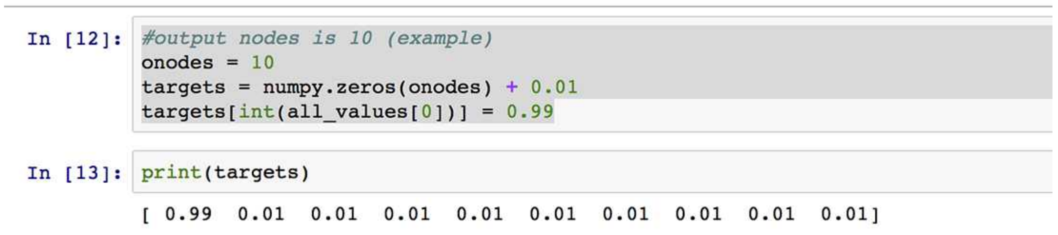
```
#output nodes is 10 (example)
onodes = 10
targets = numpy.zeros(onodes) + 0.01
targets[int(all_values[0])] = 0.99
```

The first line, other than the comment, simply sets the number of output nodes to 10, which is right for our example with ten labels.

The second line simple uses a convenient numpy function called numpy.zeros() to create an array filled with zeros. The parameter it takes is the size and shape of the array we want. Here we just want a simple one of length **onodes**, which the number of nodes on the final output layer. We add 0.01 to fix the problem with zeros we just talked about.

The next line takes the first element of the MNIST dataset record, which is the training target label, and converts that string into an integer. Remember that record is read from the source files as a text string, not a number. Once that conversion is done, that target label is used to set the right element of the targets list to 0.99. It looks neat because a label of "0" will be converted to integer 0, which is the correct index into the **targets**[] array for that label. Similarly, a label of "9" will be converted to integer 9, and **targets**[9] is indeed the last element of that array.

The following shows an example of this working:



```
In [12]: #output nodes is 10 (example)
         onodes = 10
         targets = numpy.zeros(onodes) + 0.01
         targets[int(all_values[0])] = 0.99

In [13]: print(targets)

         [ 0.99  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01]
```

Excellent, we have now worked out how to prepare the inputs for training and querying, and the outputs for training too.

Let's update our Python code to include this work. The following shows the code developed thus far. The code will always be available on github at the following link, but will evolve as we add more to it:

- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part2_neural_network_mnist_data.ipynb

You can always see the previous versions as they've developed at the following history view:

- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/commits/master/part2_neural_network_mnist_data.ipynb

```
# python notebook for Make Your Own Neural Network
# code for a 3-layer neural network, and code for learning the MNIST
dataset
# (c) Tariq Rashid, 2016
# license is GPLv2

import numpy
# scipy.special for the sigmoid function expit()
import scipy.special
# library for plotting arrays
import matplotlib.pyplot
# ensure the plots are inside this notebook, not an external window
%matplotlib inline

# neural network class definition
class neuralNetwork:


    # initialise the neural network
    def __init__(self, inputnodes, hiddennodes, outputnodes,
learningrate):
        # set number of nodes in each input, hidden, output layer
        self.inodes = inputnodes
        self.hnodes = hiddennodes
        self.onodes = outputnodes

        # link weight matrices, wih and who
        # weights inside the arrays are w_i_j, where link is from
node i to node j in the next layer
        # w11 w21
        # w12 w22 etc
        self.wih = numpy.random.normal(0.0, pow(self.hnodes, -0.5),
(self.hnodes, self.inodes))
        self.who = numpy.random.normal(0.0, pow(self.onodes, -0.5),
(self.onodes, self.hnodes))
```

```
        # learning rate
        self.lr = learningrate

        # activation function is the sigmoid function
        self.activation_function = lambda x: scipy.special.expit(x)

        pass


    # train the neural network
    def train(self, inputs_list, targets_list):
        # convert inputs list to 2d array
        inputs = numpy.array(inputs_list, ndmin=2).T
        targets = numpy.array(targets_list, ndmin=2).T

        # calculate signals into hidden layer
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calculate the signals emerging from hidden layer
        hidden_outputs = self.activation_function(hidden_inputs)

        # calculate signals into final output layer
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calculate the signals emerging from final output layer
        final_outputs = self.activation_function(final_inputs)

        # output layer error is the (target - actual)
        output_errors = targets - final_outputs
        # hidden layer error is the output_errors, split by weights,
recombined at hidden nodes
        hidden_errors = numpy.dot(self.who.T, output_errors)

        # update the weights for the links between the hidden and
output layers
        self.who += self.lr * numpy.dot((output_errors *
final_outputs * (1.0 - final_outputs)),
numpy.transpose(hidden_outputs))

        # update the weights for the links between the input and
hidden layers
        self.wih += self.lr * numpy.dot((hidden_errors *
hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))

        pass
    # query the neural network
    def query(self, inputs_list):
        # convert inputs list to 2d array
        inputs = numpy.array(inputs_list, ndmin=2).T

        # calculate signals into hidden layer
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calculate the signals emerging from hidden layer
        hidden_outputs = self.activation_function(hidden_inputs)

        # calculate signals into final output layer
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calculate the signals emerging from final output layer
        final_outputs = self.activation_function(final_inputs)

        return final_outputs

# number of input, hidden and output nodes
input_nodes = 784
hidden_nodes = 100
output_nodes = 10

# learning rate is 0.3
learning_rate = 0.3

# create instance of neural network
n = neuralNetwork(input_nodes,hidden_nodes,output_nodes,
learning_rate)

# load the mnist training data CSV file into a list
training_data_file = open("mnist_dataset/mnist_train_100.csv", 'r')
training_data_list = training_data_file.readlines()
training_data_file.close()

# train the neural network

# go through all records in the training data set
for record in training_data_list:
    # split the record by the ',' commas
    all_values = record.split(',')
    # scale and shift the inputs
    inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
    # create the target output values (all 0.01, except the desired
label which is 0.99)
    targets = numpy.zeros(output_nodes) + 0.01
    # all_values[0] is the target label for this record
    targets[int(all_values[0])] = 0.99
    n.train(inputs, targets)
    pass
```

You can see we’ve imported the plotting library at the top, added some code to set the size of the input, hidden and output layers, read the smaller MNIST training data set, and then trained the neural network with those records.

Why have we chosen 784 input nodes? Remember, that’s 28 x 28, the pixels which make up the handwritten number image.

The choice of 100 hidden nodes is not so scientific. We didn’t choose a number larger than 784 because the idea is that neural networks should find features or patterns in the input which can be expressed in a shorter form than the input itself. So by choosing a value smaller than the number of inputs, we force the network to try to summarise the key features. However if we choose too few hidden layer nodes, then we restrict the ability of the network to find sufficient features or patterns. We’d be taking away its ability to express its own understanding of the MNIST data. Given the output layer needs 10 labels, hence 10 output nodes, the choice of an intermediate 100 for the hidden layer seems to make sense.

It is worth making an important point here. There isn’t a perfect method for choosing how many hidden nodes there should be for a problem. Indeed there isn’t a perfect method for choosing the number of hidden layers either. The best approaches, for now, are to experiment until you find a good configuration for the problem you’re trying to solve.

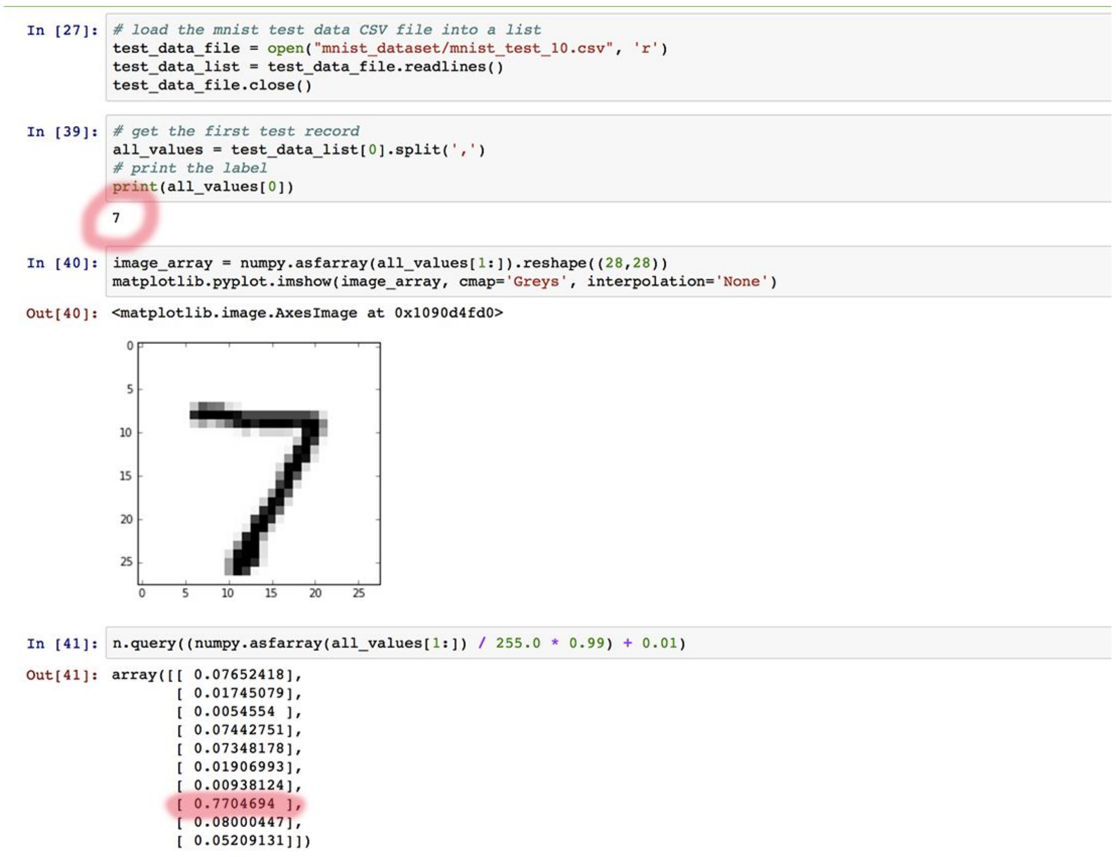
Testing the Network
Now that we’ve trained the network, at least on a small subset of 100 records, we want to test how well that worked. We do this against the second data set, the training dataset.

We first need to get at the test records, and the Python code is very similar to that used to get the training data.

```
# load the mnist test data CSV file into a list
test_data_file = open("mnist_dataset/mnist_test_10.csv", 'r')
test_data_list = test_data_file.readlines()
test_data_file.close()
```

We unpack this data in the same way as before, because it has the same structure.

Before we create a loop to go through all the test records, let’s just see what happens if we manually run one test. The following shows the first record from the test data set being used to query the now trained neural network.



```
In [27]: # load the mnist test data CSV file into a list
         test_data_file = open("mnist_dataset/mnist_test_10.csv", 'r')
         test_data_list = test_data_file.readlines()
         test_data_file.close()

In [39]: # get the first test record
         all_values = test_data_list[0].split(',')
         # print the label
         print(all_values[0])

         7

In [40]: image_array = numpy.asfarray(all_values[1:]).reshape((28,28))
         matplotlib.pyplot.imshow(image_array, cmap='Greys', interpolation='None')

Out[40]: <matplotlib.image.AxesImage at 0x1090d4fd0>
```

```
In [41]: n.query((numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01)

Out[41]: array([[ 0.07652418],
                [ 0.01745079],
                [ 0.0054554 ],
                [ 0.07442751],
                [ 0.07348178],
                [ 0.01906993],
                [ 0.00938124],
                [ 0.7704694 ],
                [ 0.08000447],
                [ 0.05209131]])
```

You can see that the label for the first record from the test data set is “7”. That’s what we hope the neural network will answer when we query it.

Plotting the pixel values as an image confirms the handwritten number is indeed a “7”.

Querying the trained network produces a list of numbers, the outputs from each of the output nodes. You can quickly see that one output value is much larger than the others, and is the one corresponding to the label “7”. That’s the eighth element, because the first one corresponds to the label “0”.

It worked!

This is real moment to savour. All our hard work throughout this guide was worth it!

We trained our neural network and we just got it to tell is what it thinks is the number represented by that picture. Remember that it hasn’t seen that picture before, it wasn’t part of the training data set. So the neural network was able to correctly classify a handwritten character that it had not seen before. That is massive!

With just a few lines of simple Python we have created a neural network that learns to do something that many people would consider artificial intelligence - it learns to recognise images of human handwriting.

This is even more impressive given that we only trained on a tiny subset of the full training data set. Remember that training data set has 60,000 records, and we only trained on 100. I personally didn't think it would work!

Let's crack on and write the code to see how the neural network performs against the rest of the data set, and keep a score so we can later see if our own ideas for improving the learning worked, and also to compare with how well others have done this.

It's easiest to look at the following code and talk through it:

```
# test the neural network

# scorecard for how well the network performs, initially empty
scorecard = []

# go through all the records in the test data set
for record in test_data_list:
    # split the record by the ',' commas
    all_values = record.split(',')
    # correct answer is first value
    correct_label = int(all_values[0])
    print(correct_label, "correct label")
    # scale and shift the inputs
    inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
    # query the network
    outputs = n.query(inputs)
    # the index of the highest value corresponds to the label
    label = numpy.argmax(outputs)
    print(label, "network's answer")
    # append correct or incorrect to list
    if (label == correct_label):
        # network's answer matches correct answer, add 1 to 
scorecard
        scorecard.append(1)
    else:
        # network's answer doesn't match correct answer, add 0 to 
scorecard
        scorecard.append(0)
        pass
        pass
```

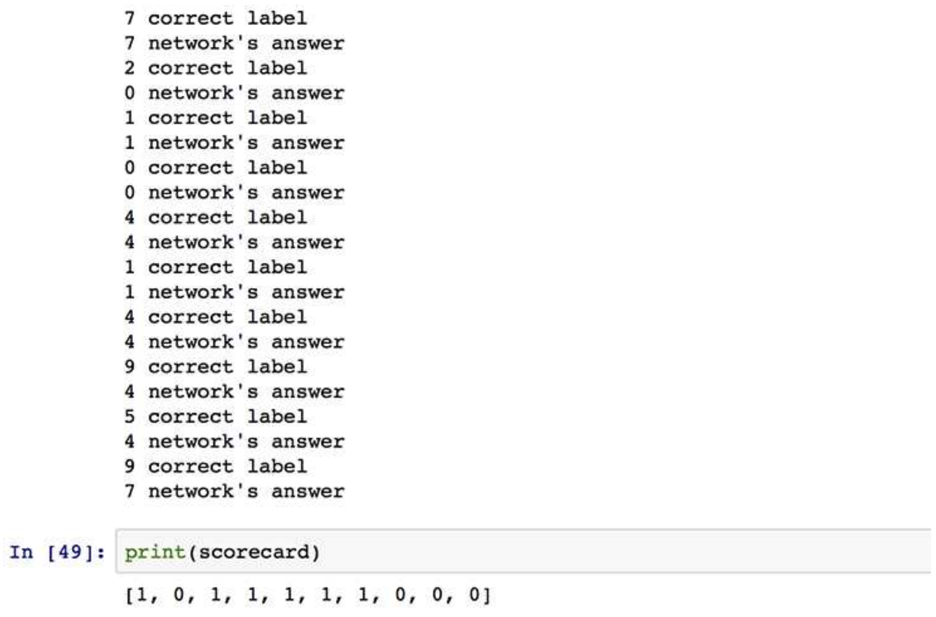
Before we jump into the loop which works through all the test data set records, we create an empty list, called **scorecard**, which will be the scorecard that we update after each record.

You can see that inside the loop, we do what we did before, we split the text record by the commas to separate out the values. We keep a note of the first value as the correct answer. We grab the remaining values and rescale them so they're suitable for querying the neural network. We keep the response from the neural network in a variable called **outputs**.

Next is the interesting bit. We know the output node with the largest value is the one the network thinks is the answer. The index of that node, that is, its position, corresponds to the label. That's a long way of saying, the first element corresponds to the label "0", and the fifth element corresponds to the label "4", and so on. Luckily there is a convenient numpy function that finds the largest value in an array and tells us its position, numpy.argmax(). You can read about it online here. If it returns 0 we know the network thinks the answer is zero, and and so on.

That last bit of code compares the label with the known correct label. If they are the same, a "1" is appended to the scorecard, otherwise a "0" is appended.

I've included some useful print() commands in the code so that we can see for ourselves the correct and predicted labels. The following shows the results of this code, and also of printing out the scorecard.



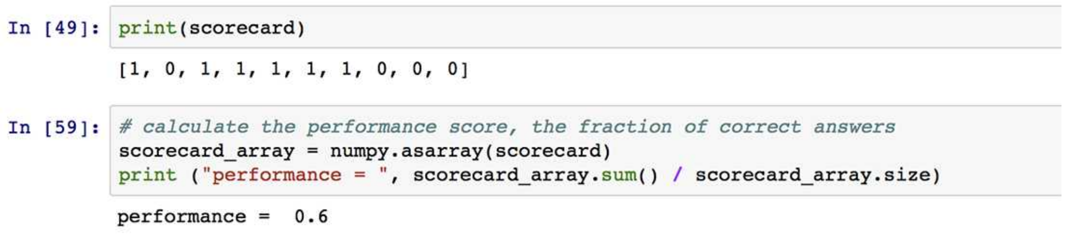
```
7 correct label
7 network's answer
2 correct label
0 network's answer
1 correct label
1 network's answer
0 correct label
0 network's answer
4 correct label
4 network's answer
1 correct label
1 network's answer
4 correct label
4 network's answer
9 correct label
4 network's answer
5 correct label
4 network's answer
9 correct label
7 network's answer

In [49]: print(scorecard)

[1, 0, 1, 1, 1, 1, 1, 0, 0, 0]
```

Not so great this time! We can see that there are quite a few mismatches. The final scorecard shows that out of ten test records, the network got 6 right. That's a score of 60%. That's not actually too bad given the small training set we used.

Let's finish off with some code to print out that test score as a fraction.

```
# calculate the performance score, the fraction of correct answers
scorecard_array = numpy.asarray(scorecard)
print ("performance = ", scorecard_array.sum() /
scorecard_array.size)
```

This is a simple calculation to workout the fraction of correct answers. It's the sum of "1" entries on the scorecard divided by the total number of entries, which is the size of the scorecard. Let's see what this produces.



```
In [49]: print(scorecard)

         [1, 0, 1, 1, 1, 1, 1, 0, 0, 0]

In [59]: # calculate the performance score, the fraction of correct answers
         scorecard_array = numpy.asarray(scorecard)
         print ("performance = ", scorecard_array.sum() / scorecard_array.size)

         performance =  0.6
```

That produces the fraction 0.6 or 60% accuracy as we expected.

Training and Testing with the Full Datasets
Let's add all this new code we've just developed to testing the network's performance to our main program.

While we're at it, let's change the file names so that we're now pointing to the full training data set of 60,000 records, and the test data set of 10,000 records. We previously saved those files as **mnist_dataset/mnist_train.csv** and **mnist_dataset/mnist_test.csv**. We're getting serious now!

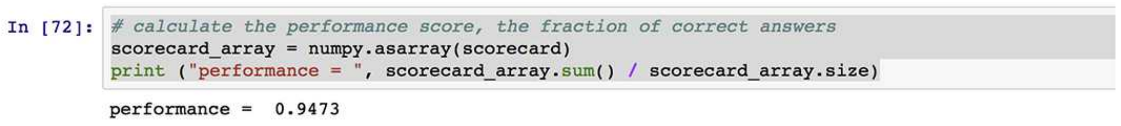
Remember, you can get the Python notebook online at github:

- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part2_neural_network_mnist_data.ipynb

The history of that code is also available on github so you can see the code as it developed:

- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/commits/master/part2_neural_network_mnist_data.ipynb

The result of training our simple 3-layer neural network against the full 60,000 training examples, and then testing it against the 10,000 records, gives us an overall performance score of **0.9473**. That is very very good. Almost 95% accurate!



```
In [72]: # calculate the performance score, the fraction of correct answers
         scorecard_array = numpy.asarray(scorecard)
         print ("performance = ", scorecard_array.sum() / scorecard_array.size)

         performance =  0.9473
```

It is worth comparing this score of just under 95% accuracy against industry benchmarks recorded at http://yann.lecun.com/exdb/mnist/. We can see that we're better than some of the historic benchmarks, we are about the same performance as the simplest neural network approach listen there, which has a performance of 95.3%.

That's not bad at all. We should be very pleased that our very first go at a simple neural network achieves the kind of performance that a professional neural network researcher achieved.

By the way, it shouldn't surprise you that crunching through 60,000 training examples, each requiring a set of feedforward calculations from 784 input nodes, through 100 hidden nodes, and also doing an error feedback and weight update, all takes a while even for a fast modern home computer. My new laptop took about 2 minutes to get through the training loop. Yours may be quicker or slower.

Some Improvements: Tweaking the Learning Rate

A 95% performance score on the MNIST dataset with our first neural neural network, using only simple ideas and simple Python is not a bad at all, and if you wanted to stop here you would be entirely justified.

But let's see if we can make some easy improvements.

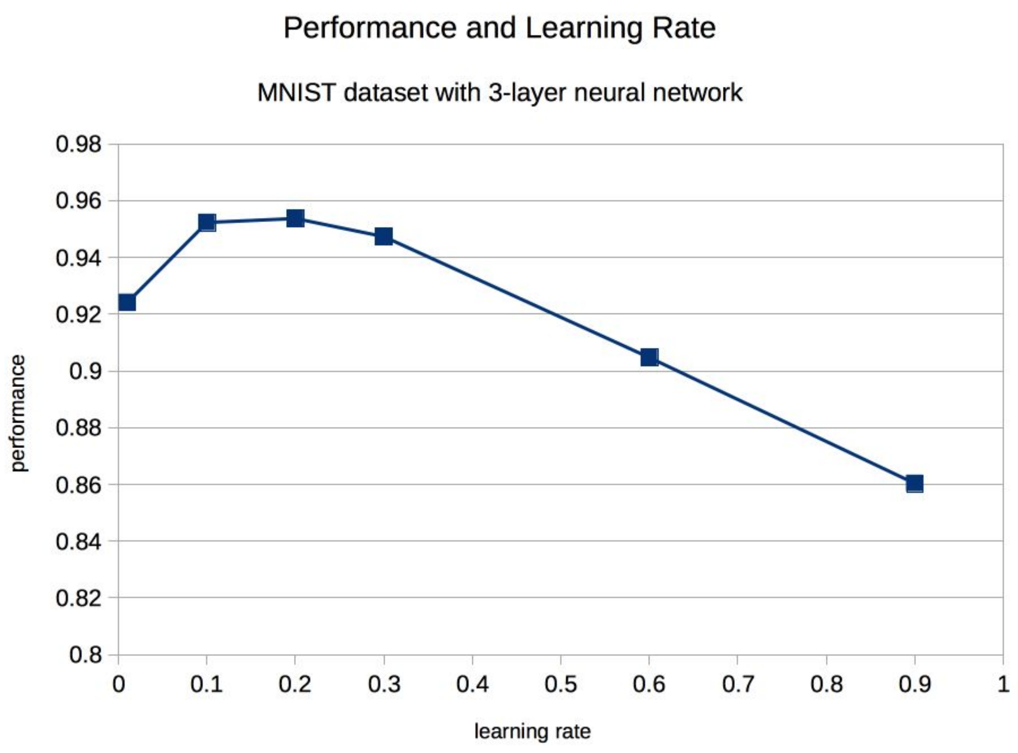
The first improvement we can try is to adjust the learning rate. We set it at 0.3 previously without really experimenting with different values.

Let's try doubling it to **0.6**, to see if a boost will actually be helpful or harmful to the overall network learning. If we run the code we get a performance score of **0.9047**. That's worse than before. So it looks like the larger learning rate leads to some bouncing around and overshooting during the gradient descent.

Let's try again with a learning rate of **0.1**. This time the performance is an improvement at **0.9523**. It's similar in performance to one listed on that website which has 1000 hidden nodes. We're doing well with much less!

What happens if we keep going and set a learning rate of an even smaller **0.01**? The performance isn't so good at **0.9241**. So it seems having too small a learning rate is damaging. This makes sense because we're limiting the speed at which gradient descent happens, we're making the steps too small.

The following plots a graph of these results. It's not a very scientific approach because we should really do these experiments many times to reduce the effect of randomness and bad journeys down the gradient descent, but it is still useful to see the general idea that there is a sweet spot for learning rate.



The plot suggested that between a learning rate of 0.1 and 0.3 there might be better performance, so let's try a learning rate of **0.2.** The performance is **0.9537**. That is indeed a tiny bit better than either 0.1 and 0.3 This idea of plotting graphs to get a better feel for what is going on is something you should consider in other scenarios too - pictures help us understand much better than a list of numbers!

So we'll stick with a learning rate of 0.2, which seems to be a sweet spot for the MNIST data set and our neural network.

By the way, if you run this code yourself, your own scores will be slightly different because the whole process is a little random. Your initial random weights won't be the same as my initial random weights, and so your own code will take a different route down the gradient descent than mine.

Some Improvements: Doing Multiple Runs

The next improvement we can do is to repeat the training several times against the data set. Some people call each run through an **epoch**. So a training session with 10 epochs means running through the entire training data set 10 times. Why would we do that? Especially if the time our computers take goes up to 10 or 20 or even 30 minutes? The reason it is worth doing is that we're helping those weights do that gradient descent by providing more chances to creep down those slopes.

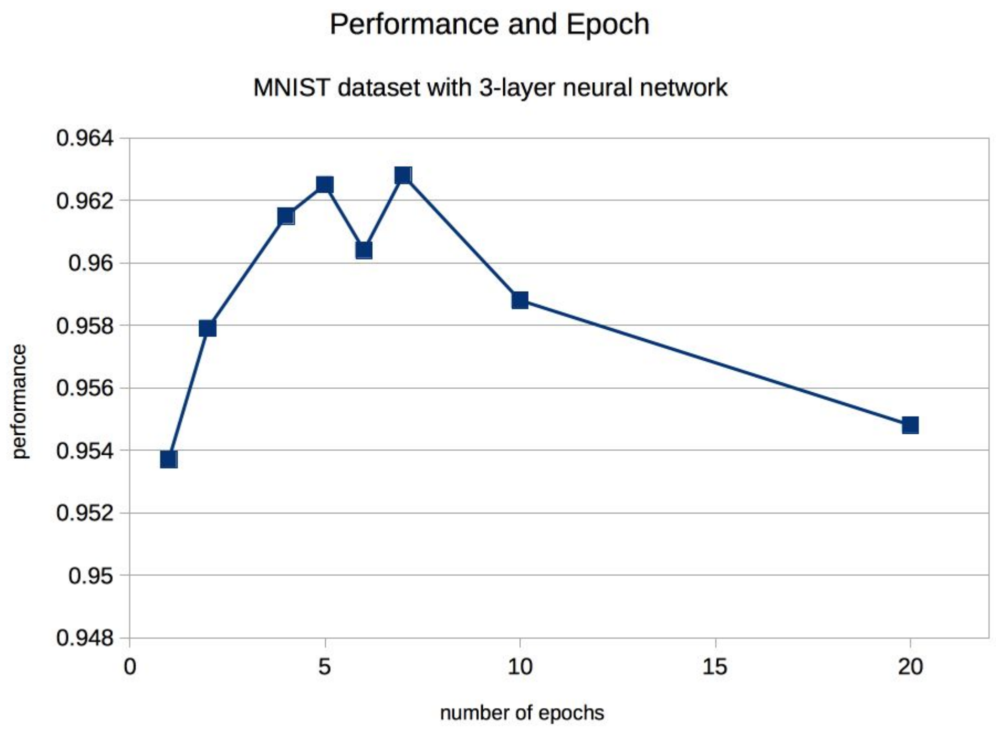
Let's try it with 2 epochs. The code changes slightly because we now add an extra loop around the training code. The following shows this outer loop colour coded to help see what's happening.

```
# train the neural network

# epochs is the number of times the training data set is used for
training
epochs = 10

for e in range(epochs):
    # go through all records in the training data set
    for record in training_data_list:
        # split the record by the ',' commas
        all_values = record.split(',')
        # scale and shift the inputs
        inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) +
0.01
        # create the target output values (all 0.01, except the
desired label which is 0.99)
        targets = numpy.zeros(output_nodes) + 0.01
        # all_values[0] is the target label for this record
        targets[int(all_values[0])] = 0.99
        n.train(inputs, targets)
        pass
    pass
```

The resulting performance with 2 epochs is **0.9579**, a small improvement over just 1 epoch.

Just like we did with tweaking the learning rate, let's experiment with a few different epochs and plot a graph to visualise the effect this has. Intuition suggests the more training you do the better the performance. Some of you will realise that too much training is actually bad because the network overfits to the training data, and then performs badly against new data that it hasn't seen before. This **overfitting** is something to beware of across many different kinds of machine learning, not just neural networks.

Here’s what happens:

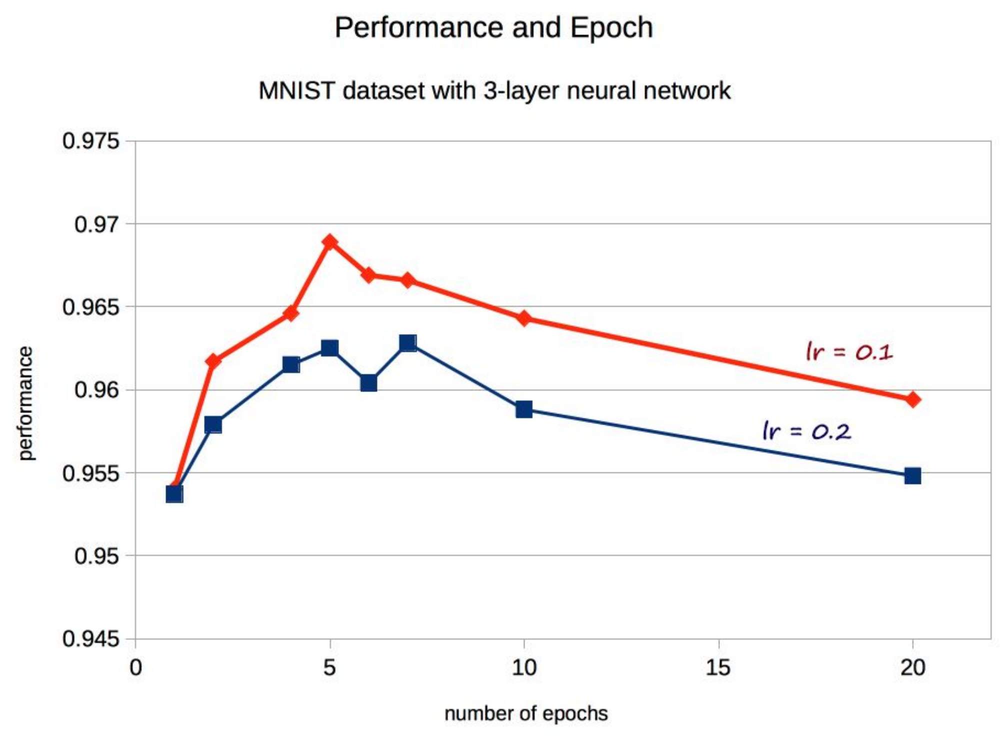


You can see the results aren’t quite so predictable. You can see there is a sweet spot around 5 or 7 epochs. After that performance degrades, and this may be the effect of overfitting. The dip at 6 epochs is probably a bad run with the network getting stuck in a bad minimum during gradient descent. Actually, I would have expected much more variation in the results because we’ve not done many experiments for each data point to reduce the effect of expected variations from what is essentially a random process. That’s why I’ve left that odd point for 6 epochs in, to remind us that neural network learning is a random process at heart and can sometimes not work so well, and sometimes work really badly.

Another idea is that the learning rate is too high for larger numbers of epochs. Lets try this experiment again and tune down the learning rate from 0.2 down to 0.1 and see what happens.

The peak performance is now up to **0.9628**, or 96.28%, with 7 epochs.

The following graph shows the new performance with learning rate at 0.1 overlaid onto the previous one.



You can see that calming down the learning rate did indeed produce better performance with more epochs. That peak of **0.9689** represents an approximate error rate of 3%, which is comparable to the networks benchmarks on Yann LeCun's website.

Intuitively it makes sense that if you plan to explore the gradient descent for much longer (more epochs), you can afford to take shorter steps (learning rate), and overall you'll find a better path down. It does seem that 5 epochs is probably the sweet spot for this neural network against this MNIST learning task. Again keep in mind that we did this in a fairly unscientific way. To do it properly you would have to do this experiment many times for each combination of learning rates and epochs to minimise the effect of randomness that is inherent in gradient descent.

Change Network Shape

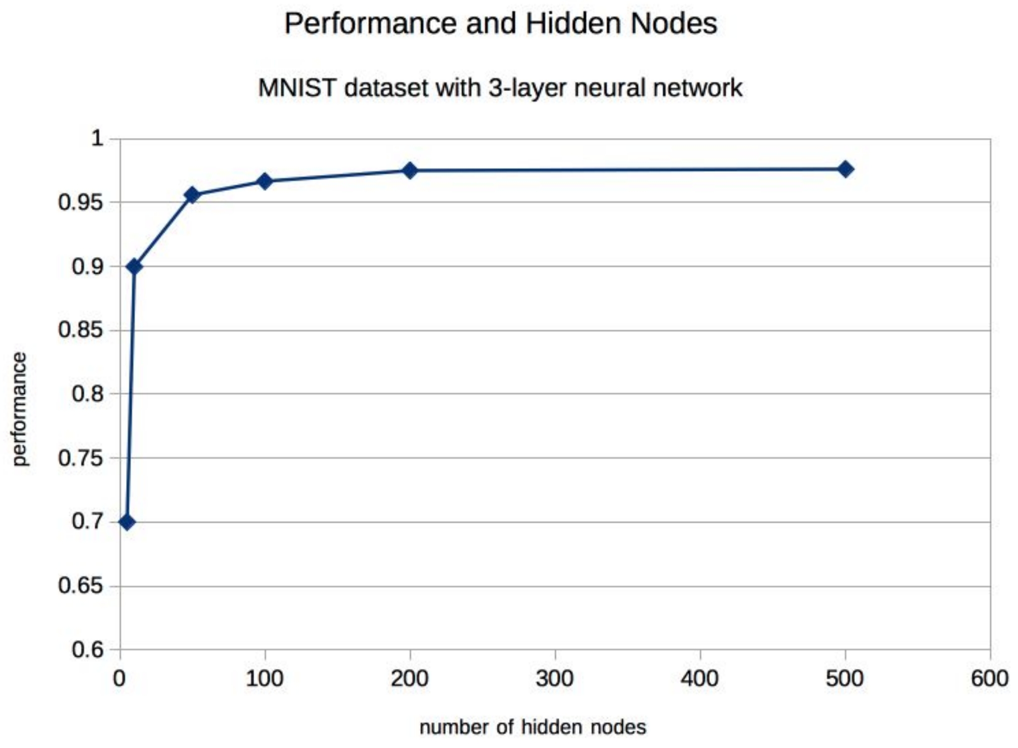
One thing we haven't yet tried, and perhaps should have earlier, is to change the shape of the neural network. Let's try changing the number of middle hidden layer nodes. We've had them set to 100 for far too long!

Before we jump in and run experiments with different numbers of hidden nodes, let's think about what might happen if we do. The hidden layer is the layer which is where the learning happens. Remember the input nodes simply bring in the input signals, and the output nodes simply push out the network's answer. It's the hidden layer (or layers) which have to learn to turn the input into the answer. It's where the learning happens. Actually, it's the link weights before and after the hidden nodes that do the learning, but you you know what I mean.

If we had too few hidden nodes, say 3, you can image there is no way there is enough space to learn whatever a network learns, to somehow turn all the inputs into the correct outputs. It would be like asking a car with 5 seats to carry 10 people. You just can't fit that much stuff inside. Computer scientists call this kind of limit a **learning capacity**. You can't learn more than the learning capacity, but you can change the vehicle, or the network shape, to increase the capacity.

What if we had 10000 hidden nodes? Well we won't be short of learning capacity, but we might find it harder to train the network because now there are too many options for where the learning should go. Maybe it would take 10000s of epochs to train such a network.

Let's run some experiments and see what happens.



You can see that for low numbers of hidden nodes the results are not as good for higher numbers. We expected that. But the performance from just 5 hidden nodes was **0.7001**. That is pretty amazing given that from such few learning locations the network is still about 70% right. Remember we've been running with 100 hidden nodes thus far. Just 10 hidden nodes gets us **0.8998** accuracy, which again is pretty impressive. That's 1/10th of the nodes we've been used to, and the network's performance jumps to 90%.

This point is worth appreciating. The neural network is able to give really good results with so few hidden nodes, or learning locations. That's a testament to their power.

As we increase the number of hidden nodes, the results do improve but not as drastically. The time taken to train the network also increases significantly, because each extra hidden node means new network links to every node in the preceding and next layers, which all require lots more calculations! So we have to choose a number of hidden nodes with a tolerable run time. For my computer that's 200 nodes. Your computer may be faster or slower.

We've also set a new record for accuracy, with **0.9751** with 200 nodes. And a long run with 500 nodes gave us **0.9762**. That's really good compared to the benchmarks listed on LeCun's website.

Looking back at the graphs you can see that the previous stubborn limit of about 95% accuracy was broken by changing the shape of the network.

Good Work!

Looking back over this work, we've created a neural network using only the simple concepts we covered earlier, and using simple Python.

And that neural network has performed so well, without any extra fancy mathematical magic, its performance is very respectable compared to networks that academics and researchers make.

There's more fun in part three of guide, but even if you don't explore those ideas, don't hesitate to experiment further with the neural network you've already made - try a different number of hidden nodes, or a different scaling, or even a different activation function, just to see what happens.

Final Code

The following shows the final code in case you can't access the code on github, or just prefer a copy here for easy reference.

```
# python notebook for Make Your Own Neural Network
# code for a 3-layer neural network, and code for learning the MNIST
dataset
# (c) Tariq Rashid, 2016
# license is GPLv2


import numpy
# scipy.special for the sigmoid function expit()
import scipy.special
```

```
# library for plotting arrays
import matplotlib.pyplot
# ensure the plots are inside this notebook, not an external window
%matplotlib inline

# neural network class definition
class neuralNetwork:


    # initialise the neural network
    def __init__(self, inputnodes, hiddennodes, outputnodes,
learningrate):
        # set number of nodes in each input, hidden, output layer
        self.inodes = inputnodes
        self.hnodes = hiddennodes
        self.onodes = outputnodes

        # link weight matrices, wih and who
        # weights inside the arrays are w_i_j, where link is from
node i to node j in the next layer
        # w11 w21
        # w12 w22 etc
        self.wih = numpy.random.normal(0.0, pow(self.hnodes, -0.5),
(self.hnodes, self.inodes))
        self.who = numpy.random.normal(0.0, pow(self.onodes, -0.5),
(self.onodes, self.hnodes))

        # learning rate
        self.lr = learningrate

        # activation function is the sigmoid function
        self.activation_function = lambda x: scipy.special.expit(x)

        pass


    # train the neural network
    def train(self, inputs_list, targets_list):
        # convert inputs list to 2d array
        inputs = numpy.array(inputs_list, ndmin=2).T
        targets = numpy.array(targets_list, ndmin=2).T

        # calculate signals into hidden layer
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calculate the signals emerging from hidden layer
        hidden_outputs = self.activation_function(hidden_inputs)

        # calculate signals into final output layer
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calculate the signals emerging from final output layer
        final_outputs = self.activation_function(final_inputs)

        # output layer error is the (target - actual)
        output_errors = targets - final_outputs
        # hidden layer error is the output_errors, split by weights,
recombined at hidden nodes
        hidden_errors = numpy.dot(self.who.T, output_errors)

        # update the weights for the links between the hidden and
output layers
        self.who += self.lr * numpy.dot((output_errors *
final_outputs * (1.0 - final_outputs)),
numpy.transpose(hidden_outputs))

        # update the weights for the links between the input and
hidden layers
        self.wih += self.lr * numpy.dot((hidden_errors *
hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))

        pass


    # query the neural network
    def query(self, inputs_list):
        # convert inputs list to 2d array
        inputs = numpy.array(inputs_list, ndmin=2).T

        # calculate signals into hidden layer
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calculate the signals emerging from hidden layer
        hidden_outputs = self.activation_function(hidden_inputs)

        # calculate signals into final output layer
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calculate the signals emerging from final output layer
        final_outputs = self.activation_function(final_inputs)

        return final_outputs
```

```
# number of input, hidden and output nodes
input_nodes = 784
hidden_nodes = 200
output_nodes = 10

# learning rate
learning_rate = 0.1


# create instance of neural network
n = neuralNetwork(input_nodes,hidden_nodes,output_nodes,
learning_rate)

# load the mnist training data CSV file into a list
training_data_file = open("mnist_dataset/mnist_train.csv", 'r')
training_data_list = training_data_file.readlines()
training_data_file.close()


# train the neural network

# epochs is the number of times the training data set is used for
training
epochs = 5

for e in range(epochs):
    # go through all records in the training data set
    for record in training_data_list:
        # split the record by the ',' commas
        all_values = record.split(',')
        # scale and shift the inputs
        inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) +
0.01
        # create the target output values (all 0.01, except the
desired label which is 0.99)
        targets = numpy.zeros(output_nodes) + 0.01
        # all_values[0] is the target label for this record
        targets[int(all_values[0])] = 0.99
        n.train(inputs, targets)
        pass
    pass


# load the mnist test data CSV file into a list
test_data_file = open("mnist_dataset/mnist_test.csv", 'r')
```

```
test_data_list = test_data_file.readlines()
test_data_file.close()


# test the neural network

# scorecard for how well the network performs, initially empty
scorecard = []

# go through all the records in the test data set
for record in test_data_list:
    # split the record by the ',' commas
    all_values = record.split(',')
    # correct answer is first value
    correct_label = int(all_values[0])
    # scale and shift the inputs
    inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
    # query the network
    outputs = n.query(inputs)
    # the index of the highest value corresponds to the label
    label = numpy.argmax(outputs)
    # append correct or incorrect to list
    if (label == correct_label):
        # network's answer matches correct answer, add 1 to
scorecard
        scorecard.append(1)
    else:
        # network's answer doesn't match correct answer, add 0 to
scorecard
        scorecard.append(0)
        pass

    pass


# calculate the performance score, the fraction of correct answers
scorecard_array = numpy.asarray(scorecard)
print ("performance = ", scorecard_array.sum() /
scorecard_array.size)
```