Using the MNIST Data Set for Deep Learning

1 Introduction

MNIST is a classic machine-learning benchmark which contains several thousand hand-drawn images of each of the 10 digits, 0 through 9.

The features of an MNIST case are simple 28 x 28 matrices of pixel intensities, while the labels are the digits 0-9. Figure 1 provides 6 of the 17230 cases from the MNIST training set. Notice the large differences between the feature matrices of two digits of the same type. A well-trained neural network can account for these differences and achieve classification success rates well over 90%.

1.1 Image Format

The raw images consist of integer pixel-intensity values in the range [0,255], organized as either arrays (i.e. a *nested* representation) or flat lists. A high intensity value indicates a high amount of foreground color (which is white in the images of Figure 1), while a low value indicates a predominance of background color (e.g., black in Figure 1).

It is often convenient (and easier to train the ANN) that these pixel values are scaled to the range [0,1] before being loaded into the neurons of your network's input layer. Here, 0 indicates a predominance of background, while 1 indicates full foreground.

2 MNIST Data Files

On the course web page, you will be provided with a .zip file containing the MNIST Data set (4 core data files) along with the file mnist_basics.py for accessing its cases. Read the comments in that file (along with the README file, provided by the original producers of the MNIST set) for more important information. Be sure to set the module variable *__mnist_path__* in mnist_basics.py (near the top of the file) to the full path to your mnist directory, where the complete contents of the .zip file should reside. When everything is installed properly, you should be able to import mnist_basics and then call the function *quicktest()*, which will produce a few digit images in separate windows on your screen.

In a nutshell, the standard reader function of MNIST files, *load_mnist*, produces numpy arrays for both images and features (which is called a *nested* representation), while mnist_basics provides functions for converting these into simple lists (called a *flat* representation) for easily interfacing with your ANN. For images, this

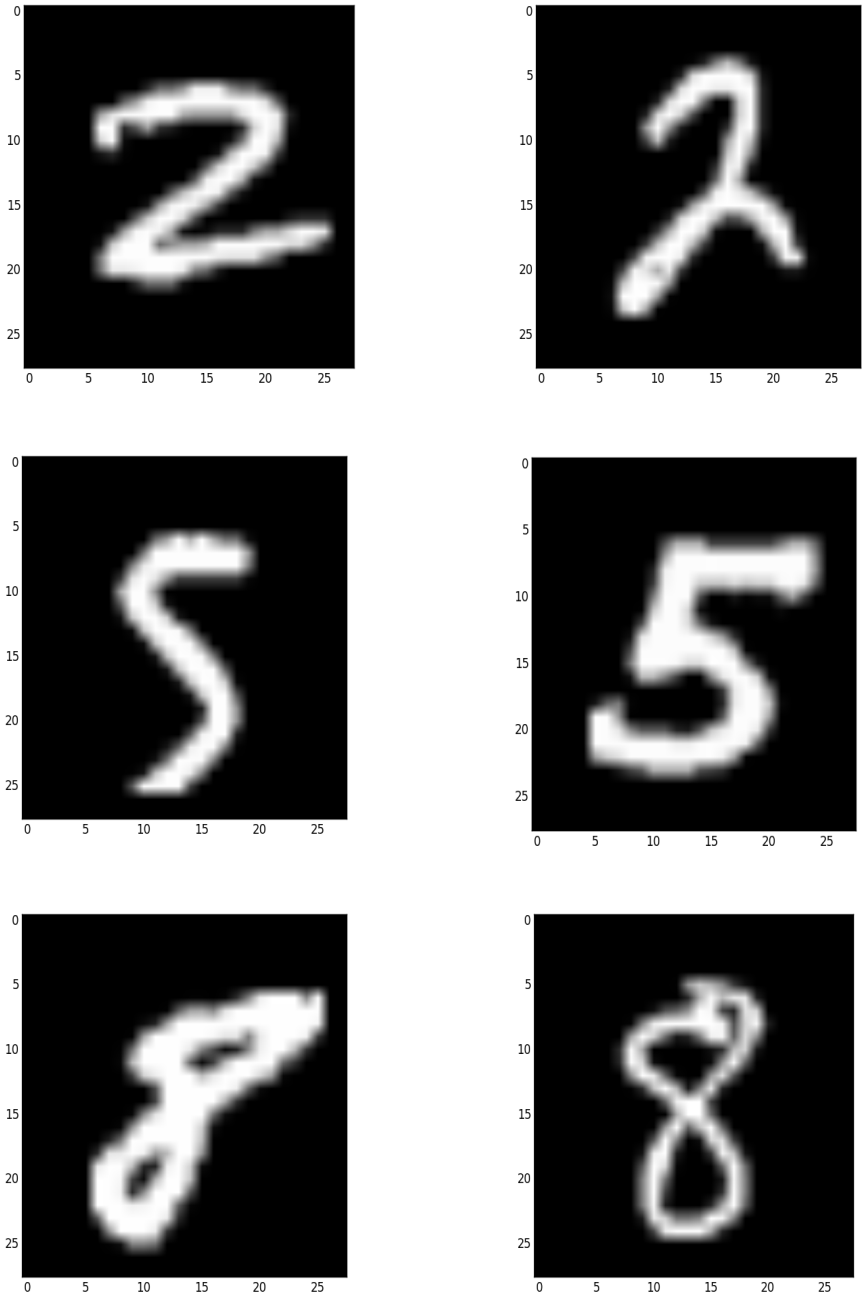

Figure 1: Sample features (28 x 28 matrices) for 3 different digits in the MNIST data set.

conversion is simply the concatenation of all rows of the array into one long vector, with the first (upper) rows appearing first in the vector. As noted earlier, these arrays consist of integers in the range [0,255].

For labels, the conversion process turns arrays like this:

[[3], [7], [2], [9], ...]

into a list of this form:

[3,7,2,9,...]

These flat representations of the entire training and test sets have been stored in two (not four) separate files, each of which contains both images and labels: *all_flat_mnist_training_cases* and *all_flat_mnist_testing_cases*. These can be loaded with the function *load_cases*, whose *nested* argument determines whether the cases should be converted to numpy arrays (when nested = True) or left as flat lists (when nested = False). When running the entire training or test sets through your ANNs, it may save time by loading these flat files directly, instead of using *load_mnist* and then having to flatten the images as preprocessing for your ANN. The only obvious reason to convert a flat representation into a nested one is for viewing the image (as discussed below).

The function *load_all_flat_cases* provides even easier access to the flattened training and test cases, but it only works on those two files, whereas *load_cases* works on any file of flattened cases. You can use the function *dump_cases* to produce your own files of flattened cases. Since these files are quite large, you may want to read in a random sample of cases from a file, as can easily be performed via slight modifications to the two file-loading functions.

To view an MNIST image in nested format, use the function *show_digit_image* in mnist_basics.py. For flat images, convert them to nested format prior to viewing via the *reconstruct_image* function.

2.1 Blind Testing on MNIST Cases

NOTE: In the Autumn 2017 version of the course IT-3105, blind testing will NOT be part of the demonstration process.

In a blind test, your system receives feature vectors but not the target labels. Your network must output predictions for those labels, which can then be checked against the (secret) target labels. To facilitate blind testing using the code in mnist_basics.py, you must:

1. Encapsulate your neural network into an object of class *ann* (or whatever name you choose).
2. Define the method *ann.blind_test(feature_sets)*.

The method *blind_test* must accept a list of sublists, where each sublist is a vector of length 784 corresponding to the raw features of one image: each sublist is a flattened image containing integers in the range [0, 255]. These raw features come directly from a flat-case file. The list does NOT contain labels, hence the adjective *blind*. Your method must produce a **flat list** of labels **predicted** by the ann when given each feature vector as input. Items in the labels list must correspond to items in the flattened image list. So if *feature_sets* consists of five image vectors, a 7, two 3's, an 8 and a 2 (in that order), then if your ann classifies them correctly, it should return this:

[7,3,3,8,2]

Note that these are **raw** features coming in as input. So if you've preprocessed your feature vectors during the training of ann (which was strongly recommended above), be sure to perform the same preprocessing on *feature_sets*.

Regardless of whether or not you use these flat-case readers while working on this project, for a blind test, you will have to be able to process a list of 120 flat cases during the demo session. To verify that your system can handle a small flat-case file, try running *load_cases* on the file named *demo100* (which contains 100 flat cases) and then entering and running those cases in your ANN via the *blind_test* method (discussed above). The cases in *demo_prep* may look strange, but a well-trained network should classify most of them correctly (though most humans probably cannot).

At the demonstration, you will import a test-driver function that will call *ann.blind_test* with different collections of image features. So even if you do not use *ann.blind_test* for anything else, make sure that it works properly prior to the demo session.