AI Programming (IT-3105) Project Module # 5:
Deep Learning for Image Classification

Purpose:

- Learn to solve classification tasks using feed-forward neural networks with backpropagation.
- Gain hands-on experience with Theano (a Python module) for configuring, training and testing neural networks

1 Assignment Overview

You will use Theano (or a system of similar capabilities) [1] to classify images of the digits 0-9 taken from the classic MNIST data set. This requires good familiarity with Theano along with clever insights into the proper topologies and parameter settings for artificial neural networks (ANNs) used as classifiers.

Your grade will be based on four factors:

- the ability to implement working ANNs,
- the successful classification of (several thousand) image examples from MNIST,
- the successful classification of several dozen additional examples derived from (but no appearing in) the MNIST collection, and
- the rigorous comparison of five different ANNs (of your design and choice).

2 Introduction

The Theano system has helped bring Deep Learning *to the masses* of students and professionals. Just a few years ago, it was difficult to build effective multi-layered ANNs for solving complex classification tasks, since deep-learning research papers normally failed to provide ALL the essential details. Today, Theano and its army of users (many of whom post their results on the web) makes the job a whole lot easier.

[1] If you choose NOT to use Theano, nor Python, please contact the course instructor very early in the project. We will need to set up a special evaluation process for you or your group.

As detailed in the lecture notes for this module, the *big wins* with Theano are both its speed in handling matrix operations, a cornerstone of Deep ANNs, and its automatic calculation of partial derivatives, often involving variables that are quite distant from one another in a sequence of mathematical expressions, e.g., the weight of an interior ANN connection and the network's final output. Together, these Theano features allow users to piece together fast ANNs using only a page or two of code.

You will build ANNs to solve a standard image classification task using the benchmark dataset known as MNIST, which contains several thousand hand-drawn images of each of the 10 digits, 0 through 9.

3 Neural Network Classifiers

ANNs are one of a standard set of tools for solving classification problems. Recently, they have begun to show exceptional performance on speech and image classification tasks, often besting humans. Human-level competence on these particular tasks has eluded artificial intelligence for many years.

Though neural networks have performed very well on many tasks for several decades, the recent quantum leap has greatly invigorated the field. It stems from many factors, including a deeper understanding of the backpropagation process, which have enabled researchers to build truly **deep** nets, often consisting of 10 or more hidden layers. Neither exceptionally deep networks nor the (somewhat complex) techniques that have sparked recent breakthroughs are the centerpiece of this project. Rather, we will focus on one tool, Theano, which serves as the computational backbone for many of these advances, but in a context that does not require very deep networks. A few hidden layers with relatively conventional dynamics (i.e., activation functions) will suffice.

However, by learning and using Theano, you will open many possibilities for further research and development using deep neural networks, whether on a masters or PhD project, or out in industry.

Building neural network classifiers is a straightforward process consisting of a few key steps:

1. Separate the data into training cases and test cases, where all cases are assumed to consist of *features* and *labels*, i.e., the correct classification.
2. Repeatedly pass the training features through the ANN to produce an output value, which yields an error term when compared to the correct classification.
3. Use error terms as the basis of backpropagation to modify weights in the network, thus *learning* to correctly classify the training cases.
4. When the total training error has been sufficiently reduced by learning, turn backpropagation off, and
5. run each test case through the ANN one time. Record the total error on the test cases and use that as an indicator of the trained ANNs ability to **generalize** to handle new cases (i.e., those that it has not explicitly trained on).

In the MNIST dataset, the features 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.

You are training the ANN to classify these matrices as the correct digit. As shown in Figure 2, the ANN should have an input layer consisting of 28 x 28 = 784 nodes, followed (downstream) by one or more hidden

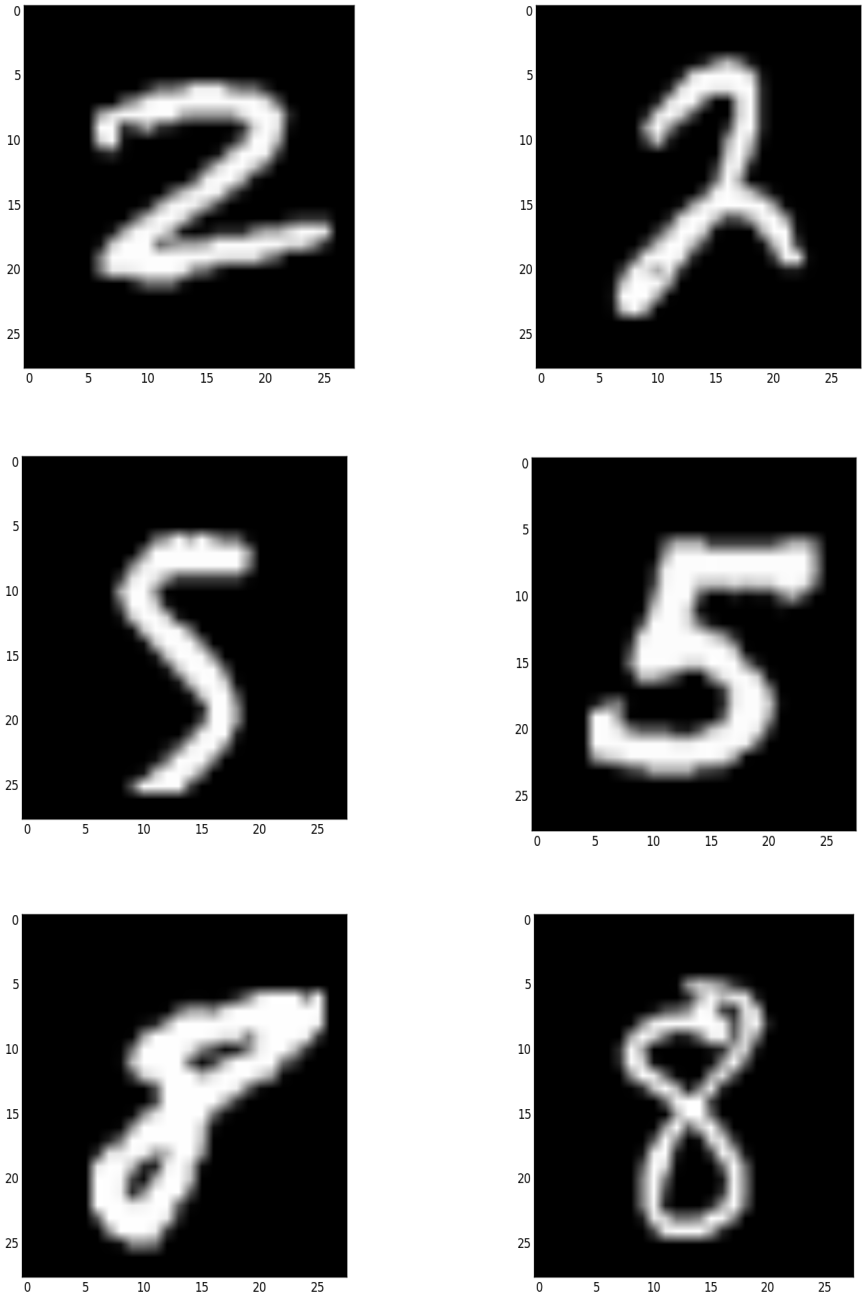

Figure 1: Sample features (28 x 28 matrices) for 3 different digits in the MNIST data set.

layers, and ending with an output layer housing 10 nodes, one for each possible digit. When features are passed through the network, the output-layer node with the highest activation should be considered the network's *answer*. Each time that answer disagrees with the correct classification, an *error* occurs.

To facilitate backpropgation, the error term for any given input matrix should be more sophisticated, measuring the sum of the errors at each output node. For this task, one output node (such as that corresponding to eight in Figure 2) should have a very high target value (such as 1.0 or 0.9), while all others should have a very low target value (such as 0 or 0.1).

3.1 Input 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).

In order to evaluate your system on demonstration day, **it is strongly advised** that these 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. Regardless of the other design decisions that you make, remember to always use this encoding for your input vector. If you train your network using any other input encoding, that network runs the risk of performing poorly during the (important) testing phase of the demonstration session.

3.2 Key Network Design Decisions

There is no magic formula for precisely determining the proper ANN for a particular task, so trial-and-error comes into play. For this assignment, some of the important degrees of freedom are:

1. The number of hidden layers.
2. The number of nodes in each hidden layer. Different layers will typically have different sizes.
3. The activation functions used in the hidden and output layers. These may even vary with the layer.
4. The learning and momentum rates used during learning.
5. The error function for backpropagation.

As mentioned earlier, this assignment should not require extremely deep networks, but anywhere from one to three hidden layers may work optimally, depending upon your choice of the other parameters. Typical activation functions with which to experiment include: sigmoid, hyperbolic tangent (tanh), and rectified linear units (RLUs). The latter have been one of the keys to the success of very deep networks. RLUs are not provided as a primitive in Theano, so you will need to code them yourself, for example, via the *theano.tensor.clip* function.

Typical error functions are the sum of squared errors and cross-entropy, which you can investigate online or in any neural network textbook.

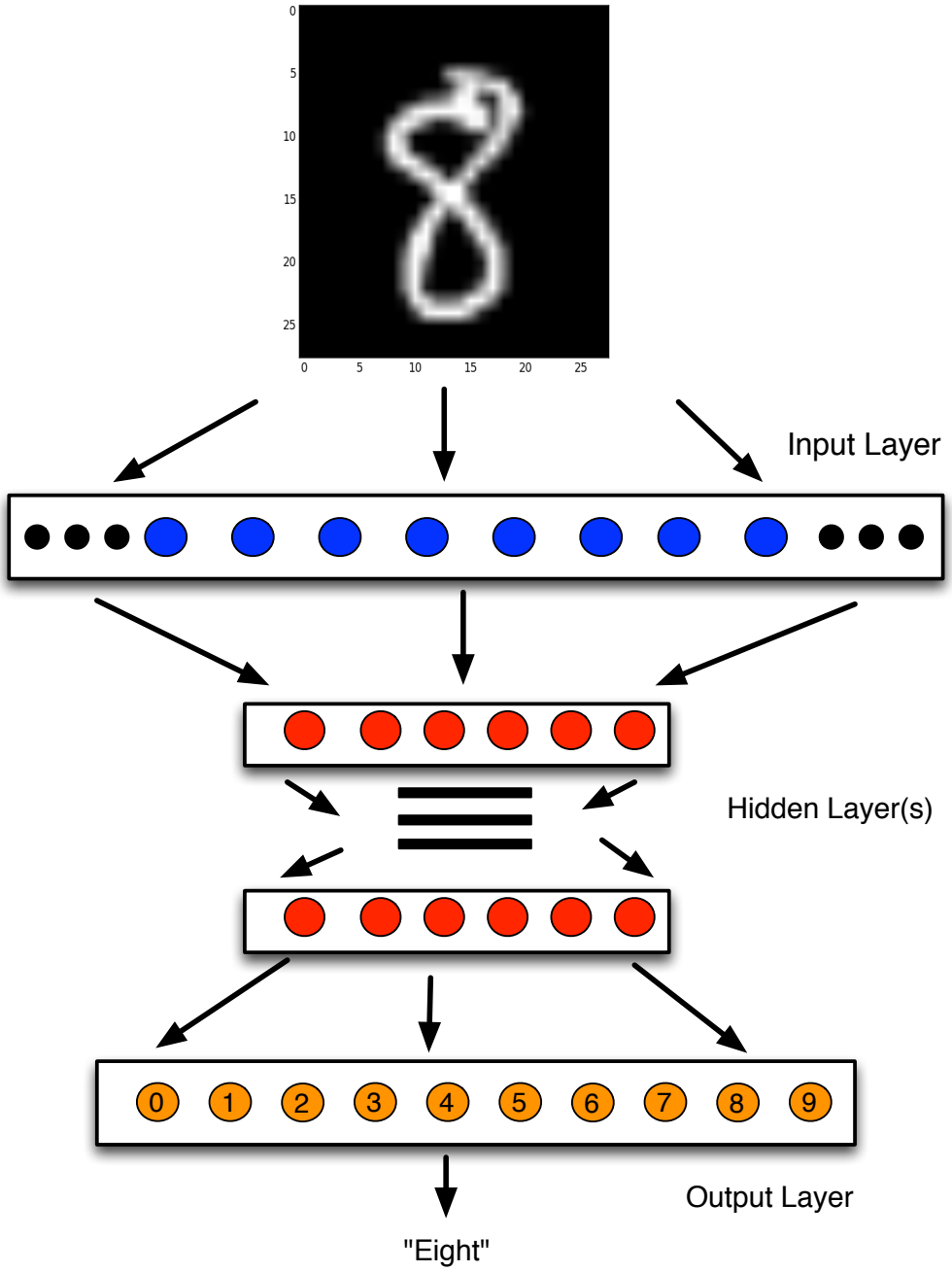

Figure 2: General framework for a neural-network classifier for the MNIST dataset.

For this assignment, you will need to design FIVE different ANNs using various combinations of the factors above. You will probably need to experiment with a lot more in order to find 5 that give somewhat useful results (i.e. reasonably low error on the training and test sets). These 5 must then be rigorously tested by running them **at least 20 times each** on the complete MNIST training and test sets. That is, for a given net, you must perform the following steps 20 times: backpropagation training followed by testing on both the original training data and the test data. All results must be recorded and summarized as part of a statistically legitimate comparison between the 5 designs. That comparison constitutes the core of the report for this module.

The best of these 5 designs should then be that which you run (i.e. train and test) at the demo session.

3.3 Automated ANN Construction

Since you will be designing and running many different ANNs, your code must facilitate the automated construction of new networks when given a basic descriptor. At the very least, your system must accept a topology specification such as [20,50,30], which indicates that three hidden layers, of sizes 20, 50 and 30 neurons, respectively, should be generated. You should not need to modify source code to try new numbers or sizes of hidden layers. For this type of ANN task, it is fine to assume that every layer (L1) is fully connected to its immediate downstream neighbor layer (L2): every neuron in L1 sends its output to every neuron in L2 along a weighted connection. No intra-layer connections nor connections to upstream or non-neighbor downstream layers are necessary (or desirable) for this task.

Your specification may include other factors, such as the learning rate, activation functions at each layer, etc., but the only strict requirement is that the basic topology of the hidden layers should be generated automatically. Failure to incorporate this minimal level of automated design will result in a loss of points on this module.

3.4 Blind Testing

In order to successfully complete the demo, you **MUST** do the following:

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*.

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.

4 The MNIST Data Set

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) 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.

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 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.

Regardless of whether or not you use these flat-case readers while working on this project, 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).

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.

5 The Demonstration Procedure

Immediately prior to the demonstration, you will be given the code file **mnistdemo.py**, along with several data files (most of which you will have received earlier). All of these files must reside in your main MNIST directory, defined by the __mnist_path__ variable. Your main code file for the MNIST module should then include:

import mnistdemo

For demonstrations of performance on the MNIST training and test sets, your system will first be trained (for as long as you choose, but within the a reasonable time limit, e.g. 1 hour) using ONLY the MNIST training-set data. After that, you will turn learning OFF in your system and test the classification accuracy of the trained network on these three data sets: MNIST training data, MNIST test data, and 120 auxiliary test cases. As described below, this testing will employ mnistdemo.py and your *blind_test* method. [2]

After training your neural net object (ann), you will submit it to testing via the following call:

mnistdemo.major_demo(ann,r,d)

where r is a number supplied to you by one of the course instructors or assistants, and d is the directory in which your MNIST data files are located. *major_demo* will then send various image lists to your neural network via the call *ann.blind_test(images)*. It will compare your network's predicted labels to the correct labels, indicate the network's accuracy, and display your point totals for each of the three test sets.

For all three sets of cases, the grading criteria (i.e., points, P) are based on the percentage of correct classifications, CC, using the following formula:

$$P = \left\lceil 5 - \frac{100 - CC}{K} \right\rceil \tag{1}$$

P will be scaled to fit within the range [0, 5], and P values will be rounded up to the nearest integer. The value of K will be 4 for the first two tests, and 8 for the third test.

6 Deliverables

1. A 3-page report (**5 points**) that:
   - Describes FIVE different ANNs that were constructed in Theano, using different combinations of hidden layers, sizes of hidden layers (in terms of the number of neurons that they contain), and activation functions.
   - Compares the training and testing performance of all FIVE ANNs, with convincing statistics used to determine which network performs best.
2. A demonstration of your best-performing ANN as it runs on:
   - The (huge) training set provided by MNIST (**5 points**)

[2] It is important that any individual or group that decides to implement a Theano-like system from scratch, but in a language other than Python, takes immediate contact with the course instructor, since this evaluation scheme is only designed for Python users.

- The (huge) test set provided by MNIST (**5 points**)
- The (small) auxiliary test set designed specifically for this demo (**5 points**), but which you will not see until the demonstration.

6.1 Other Practical Information

A zip file containing your report along with the commented code must be uploaded to It's Learning prior to the demo session in which this project module is evaluated. You will not get explicit credit for the code, but it is crucial that we have the code online in the event that you decide to register a formal complaint about your grade (for the entire course).

The 20 total points for this module are 20 of the 100 points that are available for the entire semester.

The due date for this module is the 3rd demo session.