Project 2: OCR

v20200316-1315

Due: 2 April 2020

One of the tasks often used to demonstrate and describe neural networks is that of visual recognition: how to take an image and process it to identify what it contains? In this project, you'll build networks to perform optical character recognition on digits in images.

The MNIST corpus

In the shared directory (`/home/shared/389/`) I have put a copy of the MNIST corpus of handwritten digits. This data contains one digit per 28x28 greyscale image, and the images have had some preprocessing done to resize and reposition them into a standard layout. The directory has a link to further information about the data, including its source and some of the high-performance OCR models that have been published to classify it, but the most important part of that link is the one that tells you the layout of the data files.

I will not fully recap it here, but note that each data file contains thousands of images, in binary format. You will need to do a little work to read in the data, before you can really start applying any neural networks.

The other thing to note is that the data is already split for us into training and testing. We will follow the standard split: use the 60k training images to build your model, and then the 10k test images to evaluate its performance.

Checkpoint

As usual, there's a bit of grunt work to front-load before we can get to the fun stuff. For the checkpoint, I want you to write a program that is capable of processing the MNIST data files and producing image files to show what the data looks like; your program should take three command-line arguments to identify the index of the image you want to extract, the file containing the images, and the file containing the labels (in that order). It should produce a file in the PGM format (see below) that shows the visual representation of that image from the file.

That is,
1) the index,
2) the image file,
3) the label file

For the input file format, see the link in the mnist directory; and note that for the data stored as 32-bit integer that the endianness of the data files is the opposite of that used on our linux systems, so you'll need to use a byteswap function or otherwise deal with that. (Or, just ignore them. The image files have exactly 16 bytes of header and the index files have 8 bytes of header, and you can safely skip them unless you're doing something fancy involving the number of items to read.)

Because graphics per se are not the overall focus here, we'll be using a graphics format called PGM, which isn't very efficient but is *very* easy to read and write. In fact, we'll process a subset of it, whose files begin as follows:

```
P2
#Comment about the file
28 28
255
```

The `P2` identifies it as a PGM file in ASCII format; the other three numbers identify, respectively, the width, the height, and the maximum grey level for a pixel (which ranges from 0–255). After this header, there follows a whitespace-separated list of greyscale values, one per pixel, in row-major order. The comment should include the name of the source file, the index number, and the correct label for this image.

Most standard Linux image viewers (such as `eog` or `xviewer`) can display this format.[1]

Your checkpoint program can dump this file to a hardcoded name (like `img.pgm`), or to standard input, or take a fourth command line parameter for purposes of output file naming. Your documentation should say which, so I can test it.

You will probably want to keep this program around intact as you continue on to the main project, as it will let you inspect some of the training data that you're working with; but you'll want to build the main project as a separate executable. So, you might want to pull some of this into separate functions/methods in a separate file.

The checkpoint is due **Thursday, 19 March at 11:59pm**. (FIXED)

[1] You can also use `convert`, as in

```
convert imagename.pgm imagename.png
```

to turn it into something that basically any image program can look at.

The goal

Your final version will read the training set and use it to train a multilayer neural network to classify images according to which numeric digit they contain. It will then read the test set, classify its images and compare with the correct answer, and report its results. Finally, you will implement some aspect of this in a parameterised way (for instance: how many perceptrons in the hidden layer?) and compare performance with different values of that parameter. The parameter must be somehow changeable from the command line (i.e. not require recompiling in order to change it).

A *vital part* of your program's output will be reporting on what's going on internally (i.e. your program can't just output a single final performance percentage!), but its exact format is largely up to you.

Note that once you've got the checkpoint done (and have seen how easy it is to generate PGM files), producing an *image* as a visual representation of a vector of weights may be the most effective way to understand what the perceptrons are training up to: as long as you scale your numbers to the range $[0, 255]$ (and print them as ints), the image should work, and will show you which input pixels a particular neuron is focusing on.

Final version

A full-credit final version will be a complete, non-buggy, working implementation of a noisy-channel typo correction system, TOGETHER WITH convincing proof that it is correct. The "proof" should consist of test cases (in whatever format is convenient to you) to illustrate various situations, including both input and expected results.

Remember that there need to be clear instructions on how to run it in general as well as how to run each/all of the tests and quickly verify that they ran correctly (and which rubric items each one corresponds to); and don't forget to explain how to enter actions and interpret the display! Having complete and correct documentation is an easy 15 points, but if your documentation omits important info or tells me the wrong thing, you'll get less than full credit there.

After checkpoint work (15 points) and documentation (15 points), there remain 70 points in the rubric, which will be awarded according to the

table below. Note that number of points does *not* necessarily correspond to difficulty; and you should probably implement the starred items before you move on to any other part of the implementation.

Rubric

Score	Description
	Input handling:
10	Reads at least one training image from a file (per checkpoint); can generate a .pgm file representing image data (per checkpoint) and/or weight data. (Can be part of a separate executable in the final handin but if the "main" neural net program can't even read the training file it will come off this point.) *
5	Reads in and stores all training data for further processing. *
	Simple perceptrons:
10	Makes multiple passes through the training data, trains at least a single perceptron to classify images as "is a three" or "not a three". *
10	Trains a complete output layer of ten perceptrons, each associated with one digit, and can report the "winner" for any given input. **
	Multilayer neural network
6	Feeds image data forward through at least one hidden layer to output layer, classifying image accordingly.
3	Performs any substantial backpropagation of error to adjust weights on neurons in hidden layer.
3	Backpropagates error to update hidden-layer weights according to the algorithm discussed in class and in the textbook.

Score	Description
	Classification and results
5	Reads at least one image not from the training, and classifies it. *
3	Reads all images from a test file and classifies them. *
10	...and compares classification of each with its correct label and reports results. Accuracy should be reported on both training and testing sets. **
5	Meaningfully compares results from different system configurations, and reports and interprets the differences. This report can be in its own separate file but the README should clearly indicate where to find it.

* Try getting these points done first. They add to 33 rubric points, which on top of checkpoint and documentation is 63/C.

** These next; 53 makes 83/B+.

Handing in

The checkpoint and the final version are due at 11:59pm on their respective due dates. Hand them in as `proj2` using the handin script. (FIXED)

Don't forget documentation! And appropriate output that will help me understand how you think you're meeting the rubric requirements.