CS181 Assignment 2: Neural Networks

Professor David Parkes

Out Tuesday February 15th
Due Noon Friday February 25th

General Instructions:

You may work with one other person on this assignment (or you may work alone if you prefer). Each group should turn in one writeup. This assignment consists of a theoretical component and an experimental component.

In this assignment, you will design and implement a handwritten digit recognizer, which learns how to automatically classify a digit. The set of handwritten characters are supplied for you, and they come from the post office in Buffalo, NY. The ultimate goal of the classifier is to get 100% of the test set characters correctly classified as the appropriate digit, but the problem with real-world data is that it is noisy. Some of these digits could not even be pre-classified by people, so getting as close as possible to 100% is a more feasible goal.

The dataset comes from a large database (over 60k) of handwritten digits known as MNIST. There is a relevant webpage on the data at

`http://yann.lecun.com/exdb/mnist/`

We have created our own subset of this data by randomly sampling 10k of the cleaner training data and 1k of the cleaner testing data.

Each instance consists of an image, which is a 14×14 array of pixels, each of which has a value ranging from 0 to 255 representing the pixel intensity, and a label, which ranges from 0 to 9. We have provided support code in Python.

Also, for this assignment, we have created separate training, validation, and test sets for you. You will train on the training set, tune parameters by examining performance of the trained networks on the validation set, and then finally testing performance using the test set. We will not be using cross-validation for this problem set. Rather, after training and deciding on your network, you will report a single number, which is performance of the trained network on the test set.

You will find the following files in `http://www.seas.harvard.edu/courses/cs181/docs/asst2.tar.gz` (also available in .zip format):

- `training-9k.txt`: The training images.

- `validation-1k.txt`: The validation images.
- `test-1k.txt`: The test images.
- `nn.py`: Declarations for the neural network code that you will write.
- `testnn.py`: Unittests for the neural network code (you will not need to modify this file, but the tests should pass when your code is working).

Submission Instructions: Please submit your writeup for all problems in a file named `writeup.text`. Also, include the file `nn.py` in your submission. No other files are required in your submission. The instructions for using the submit script can be found here: `http://www.seas.harvard.edu/courses/cs181/how-to-submit.html`.

Problem 1

[**9 Points**] Consider training a single perceptron with the perceptron activation rule to recognize features of images. For this exercise, assume that an image is a three by three array of pixels, with each pixel being on or off. For each of the following features, either present a perceptron that recognizes the feature, or prove that no such perceptron exists.

1. **bright-or-dark** — At least 75% of the pixels are on, or at least 75% of the pixels are off.
2. **top-bright** — A larger fraction of pixels is on in the top row than in the bottom two rows.
3. **connected** — The set of pixels that are on is connected. (In technical terms, this means that if we define a graph in which the vertices are the pixels that are on, and there is an edge between two pixels if they are adjacent vertically or horizontally, then there is a path between every pair of vertices in the graph.)

Problem 2

[**12 Points**] In this problem, consider four different possible learning algorithms for the digits classification problem:

- Decision trees
- Boosted decision stumps
- Perceptrons

- Multi-layer feed-forward neural networks

One might object that decision trees and boosted decision stumps are designed for discrete attributes, and our attributes are better treated as continuous. In fact, decision trees and related algorithms can easily be modified to handle continuous attributes. For any continuous attribute X, one can consider splits of the form $X > \alpha$, where α is a real number. Of course, the learning algorithm needs to be modified to quickly find the best split point α for a potential splitting. Also, it can now make sense to split multiple times on the same attribute in a path, with different split points. The modifications are easy, so we will assume that we are given a continuous decision tree and decision stumps algorithm. One possibility is C4.5, which is a well-known implementation of decision trees that makes use of continuous attributes.

Taking into account what you have learned about the various learning algorithms as well as the domain of digit recognition, argue for or against each of the candidate approaches for this task.

Problem 3

[**70 Points**] For this problem, you will implement a neural network for the digits classification problem and perform various tasks using your implementation. Questions you will need to answer in your writeup are marked with a double arrow:

$\Rightarrow$ Do you really think you need to answer this example question in your writeup?

1. [**40 Points**] **Implementation**

   Fill in the missing functions in `nn.py` to create a working implementation of neural networks. We have provided the signatures for functions you should populate in `nn.py`. The signatures provide a skeleton for implementing a subset of the neural network structures that we have covered in class. In particular, in your implementation, you will only need to support hidden layers whose units are fully connected to units in the previous and next layers.

   The code is structured as follows: there are `Perceptron`, `NeuralNetLayer`, and `NeuralNet` classes. The `Perceptron` class contains information about a single unit in the network. This includes the current weights for incoming edges as well as the current bias weight. The `NeuralNetLayer` class contains information about a single layer (this can be an input, hidden, or output layer). A layer can be thought of as a list of perceptrons along with a list of the inputs to those perceptrons. The `NeuralNet` class is a list of layers.

   These classes are placeholders for the data in the network, but most functionality will be implemented in standalone functions that are not part of any class. Here is a high-level overview of the different types of functions you will implement:

- **Warmup:** implement the `sigmoid` function, run its unit test, and then try out the *Plot a Sigmoid Curve* task. You won't be able to run any other tasks until all your network functionality is complete, but please do run your *tests* frequently.
- **Basic math functions:** (`hidden_error`, `output_error`, `compute_delta`, `update_weight`).
- **Functions associated with a single unit or layer in the network:** (`update_pcpt`, `update_layer`, `pcpt_activation`, `feed_forward_layer`, `layer_deltas`,
`hidden_layer_error`)
- **Functions that operate on the entire network** (`build_layer_inputs_and_outputs`, `update_net`, `init_net`)
- **Functions that handle encoding issues** (`binary_encode_label`, `distributed_encode_label`, `binary_decode_net_output`, `distributed_decode_net_output`)

2. [**5 Points**] **XOR**

   Once you have implemented all of the above, your next task will be to solve XOR. In the Tasks pane, run *Solve XOR Part I* and *Solve XOR Part II.* In part II, verify that your network is correctly classifying the four instances in the problem. Then, examine the network displayed in the pane for part I. Red edges denote negative weights, and black edges denote positive weights. The thicker the edge, the greater the absolute value of the weight

   ⇒ Include a picture of the network generated in part I in your writeup. Explain how the combination of weights emulates the logical operations needed to express XOR.

   Now re-run *Solve XOR Part II* 10-15 times. Does it correctly learn XOR every time?

   ⇒ What are some reasons that your networks could intermittently fail to learn XOR? What parameters can you tune to alleviate this type of problem in general?

3. [**15 Points**] **Experimental Analysis**

   For the experimental section, define *performance* over a set of examples as (number of examples classified correctly) / (number of total examples), and *error* as 1 - *performance.*

   You should get a feel for the performance of your network by running some experiments. You can begin by running the tasks provided in the web interface. Be aware that these may take several minutes to run. You should then conduct experiments by running `nn.py` on the command line. You can type `python nn.py --help` for a list of available parameters. It may also be a good idea to look at the `main` and `experiment` methods to get a better sense for what the command line options control. Experiment with a neural network that connects inputs to outputs, but does not have any hidden units. Through your experimentation, you should identify an effective learning rate (something within an order of magnitude or two of optimal is acceptable). In your writeup,

(a) ⇒ Provide the learning rate you used. How did you select this rate?

(b) ⇒ For this learning rate, chart the training set and validation set *error* against the number of epochs from 1 to 100.

   i. ⇒ Are we in danger of overfitting by training for too many epochs?

   ii. ⇒ What is a good number of epochs to train for?

   iii. ⇒ Why is it important that we use a validation set (rather than the actual test set) to tune the number of epochs?

(c) ⇒ What is the training, validation, and test *performance* of the network trained with your chosen learning rate and number of epochs?

4. **[10 Points] Model Selection**

   NOTE: The experiments for this portion of the assignment may take a while to run, so start early!

   Experiment with networks with a single hidden layer. Try single hidden layers with 15 and 30 fully connected units in the hidden layer. You may find it useful to use the corresponding tasks in the Tasks pane.

   (a) ⇒ Experiment with the distributed and binary encoding for both 15 and 30 hidden layers. Which encoding performs better? Is this in line with what we know about the encodings? Use the better encoding for the rest of this problem.

   (b) ⇒ Provide the learning rates you used for training 15 and 30 hidden units (they can be different).

   (c) In the previous problem, we determined the number of epochs to use based on analyzing the graph of error against the number of epochs. Devise an automatic way (something you can code up) that determines when to stop training and describe it. Implement your idea. Look for a `TODO(CS181 Student)` comment in the `experiment` function in `nn.py`. (Hint: use the validation set.)

   To activate your stopping condition, run with the `--enable-stopping` switch on the command line.

   (d) ⇒ In three sentences or less, explain your stopping condition.

(e) For both 15 and 30 hidden units, use your algorithm to determine when to stop.

$\Rightarrow$ Graph training set and validation set *performance* against the number of epochs you trained for.

(f) $\Rightarrow$ How many epochs did you use for 15 hidden units? For 30 hidden units?

(g) $\Rightarrow$ What is the test performance for 15 and 30 hidden units? Does this agree with the validation set performance you observed? Which network structure (15 hidden units, 30 hidden units, no hidden units) would you choose based on these experiments?

Problem 4

[**10 Points**] As we saw, the back-propagation algorithm for neural networks performs gradient descent on the error function:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{d=1}^{n}\sum_{k=1}^{K}(y_{dk} - a_{dk})^2$$

where there are n examples $(\mathbf{x_d}, y_d)$ and K output units, so that $y_d \in [0,1]^K$. Note that we index examples by d here, to avoid any confusion of (i,j,k) indexing for units in the neural network.

Consider an alternative error function:

$$F(\mathbf{w}) = \frac{1}{2}\sum_{d=1}^{n}\sum_{k=1}^{K}(y_{dk} - a_{dk})^2 + \gamma\sum_{j\in V}\sum_{i\in \text{Parents}(j)} w_{ji}^2,$$

where the sum in the latter term is taken over all pairs of units i, j such that there is an edge from i to j.

1. [**5 Points**] Explain why you might want to use the error function F. What problem is it trying to address, and how does it address it?

2. [**5 Points**] Use gradient descent to derive a weight update rule for the error function F. In order to simplify the description of your weight update rules, you may assume the following form for $\frac{\partial E}{\partial w_{ji}}$:

$$\frac{\partial E}{\partial w_{ji}} = -\sum_{d=1}^{n} a_{id}\delta_{jd},$$

where a_{id} is activation of unit i for example d and δ_{jd} is the back propagation error at unit j given example d. The weight update rule for weight w_{ji} should be expressed in terms of $a_{id}, \delta_{jd}, w_{ji}$, the learning rate α, and the parameter γ.