CS181 Assignment 2: Neural Networks

Out Monday February 18th
Due Friday March 1st

Submit by **noon** via iSites dropbox.

General Instructions

You may work with **one other person** on this assignment. Each pair 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 webpage about 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. The `README` file contains more thorough descriptions of the code we have distributed.

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 examine 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/files/hw2.tar.gz`:

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

- `validation-1k.txt`: The validation images.
- `test-1k.txt`: The test images.
- `data_reader.py`: Declarations and implementations for the image data structure and functions for loading the data.
- `neural_net.py`: Core neural network data structures. You need read the file thoroughly, but you should not need to modify it.
- `neural_net_impl.py`: Template for algorithms that need to be written.
- `neural_net_main.py`: Code that wraps the network and performs training with given network structures.
- `lecun-90c.pdf`: A paper on using neural networks for digit recognition.

0.0.1 Submission Instructions

Please submit writeups in PDF format.

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_k, one can consider splits of the form $x_k > \gamma$, where γ is a real number. Of course, the learning algorithm needs to be modified to quickly find the best split point γ 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

[68 Points] For this problem, you will implement a neural network for the digits classification problem. We have given you some basic data structures to work with in `neural_net.py`, and you will need to fill in the functions in `neural_net_impl.py`. The script `neural_net_main.py` will allow you to run the entire digit classifier. The `README` file and the class and function stubs in `neural_net_impl.py` contain further details and informations about the code. You might want to look ahead at question 4, as it might help you understand the structure of the code, and make it easier to code up question 1, 2 and 3.

Warning: The algorithms in this problem take a long time to run, so you should start early to make sure you have enough time to run all the simulations!

1. **[5 Points]** Fill in the `FeedForward` function that takes a neural network and inputs and propagates the inputs through the network. The function should be general enough for multi-layer hidden networks.

2. **[10 Points]** Fill in the `Backprop` function, that takes a neural network, inputs, target outputs, and a learning rate, and runs a single phase of back-propagation using the given image and the learning rate. The function should be general enough for multi-layer hidden networks.

3. **[5 Points]** Fill in the `Train` function, that takes a neural network, a training set of inputs and target outputs, a constant learning rate, and a number of epochs. The

function should train the network by running through the training set for the given number of epochs using the given learning rate.

4. **[10 Points]** Having written the core neural network framework, modify the `SimpleNetwork` class in `neural_net_impl.py` for the digits classification setting. For this step, use a *distributed* output encoding which creates a separate output for each digit (see source code for further details)

   (a) Modify the `EncodeLabel` function to use the distributed output encoding for a label.

   (b) Modify the `GetNetworkLabel` function that translates from the outputs in a neural network back to the target label (decodes the distributed encoding). For a given set of outputs corresponding to each digit, choose the digit with the largest output value.

   (c) Modify the `Convert` function that takes an `Image` and converts it into input values for the neural network. For this conversion, divide each pixel value by 256.0 so the values are between 0 and 1.

   (d) Write a function that initializes the weights in a network to random weights in $[-0.01, 0.01]$.

5. **[3 Points]** Why would we want to normalize the input values from $[0, 255]$ to between 0 and 1?

6. **[10 Points]** 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*. Implement a simple network that connects each input to all of the outputs, with no hidden layers. This is just a perceptron for each digit. Train the network for 100 epochs, initializing the weights to be small random weights in $[-0.01, 0.01]$. Track the training performance over each round for different fixed learning rates. Try rates with different orders of magnitude, like 1.0, 0.1, 0.01, 0.001, but do *not do* an exhaustive search in this space. Choose a learning rate that exhibits good training performance and fix it.

   (a) Provide the learning rate you used.

   (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?

7. **[10 Points]** *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 (Use the `HiddenNetwork` template). As in the previous problem, find learning rates that are appropriate for each of these configurations.

   (a) Provide the learning rates you used (they can be different for each number of hidden units).

   (b) 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. (Hint: use the validation set. Note: your stopping condition should also take a maximum epochs to train for so that we don't train forever).

   (c) For both 15 and 30 hidden units, use your algorithm to determine when to stop. Graph training set and validation set *error* against the number of epochs you trained for.

   (d) How many epochs did you use for 15 hidden units? For 30 hidden units?

   (e) Which network structure (15 or 30 hidden units) would you choose based on these experiments? Justify your answer.

   (f) What is the test set *performance* of the network you chose? How does this compare to the committee of perceptrons?

8. **[15 Points]** Using the infrastructure you have built up, devise a network structure on which you would like to run an additional experiment (use the `CustomNetwork` template). Perhaps it is something you think will outperform the best structure so far, or perhaps it is something you think should not outperform.

   You may (but are by no means required to), for example, consider alternate output encodings to the distributed encoding. (If you are in search of ideas, the paper by LeCun et al. (`lecun-90c.pdf`) describes a research group's experience building neural networks for the digits classification problem. Take away high level ideas from the paper to implement your neural network, but do not try to exactly replicate their work as their networks are very complex and are not fully specified). If your structure has some parameters that need to be tuned (e.g. the number of hidden units), consider using the validation set performance to make this choice.

   (a) Describe the network you chose to implement. This description need not be at the node by node level, but it should provide an overall view of the layers and units you created. Explain why you decided to run an experiment with this network structure.

(b) Train your network using the training and validation sets, and compare the trained network's test performance to the previous structures. Try to explain what you find. (You will not be evaluated on the absolute performance of your network.).

Problem 4

[10 Points] As we have seen, the back-propagation algorithm for neural networks performs gradient descent on the loss function:

$$\mathcal{L}(\boldsymbol{w}\,;\,\mathcal{D}) = \sum_{n=1}^{N}\sum_{j=1}^{J}(y_{nj} - a_{nj})^2$$

where there are N examples $\{\boldsymbol{x}_n, \boldsymbol{y}_n\}_{n=1}^{N}$ and J output units. The activation a_{nj} is the value of output unit j on example n. Consider an alternative error function:

$$C(\boldsymbol{w}) = \sum_{n=1}^{N}\sum_{j=1}^{J}(y_{nj} - a_{nj})^2 + \lambda\sum_{m=1}^{M}\left(\sum_{k=0}^{K} w_{km}^2 + \sum_{j=1}^{J} w_{mj}^2\right)$$

where this sums over all of the weights in a network with K inputs, a single layer of M hidden units, and J outputs.

1. **[5 Points]** Explain why you might want to use the error function C. 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 C.